RADC-TR-90-404, Vol XI (of 18)
Final Technical Report
December 1990

# KNOWLEDGE BASE MAINTENANCE

Northeast Artificial Intelligence Consortium (NAIC)

Kenneth Bowen

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

**This effort was funded partially by the Laboratory Director's fund.**

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-90-404, Volume XI (of 18) has been reviewed and is approved for publication.

APPROVED: John J. Crowter

JOHN J. CROWTER
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: R Raposo

RONALD RAPOSO
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC ( COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188



| 1. AGENCY USE ONLY (Leave Blank) | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | December 1990 | Final Sep 84 - Dec 89 |

**4. TITLE AND SUBTITLE**
KNOWLEDGE BASE MAINTENANCE

**5. FUNDING NUMBERS**
C - F30602-85-C-0008
PE - 62702F
PR - 5581
TA - 27
WU - 13
(See reverse)

**6. AUTHOR(S)**
Kenneth Bowen

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Northeast Artificial Intelligence Consortium (NAIC)
Science & Technology Center, Rm 2-296
111 College Place, Syracuse University
Syracuse NY 13244-4100

**8. PERFORMING ORGANIZATION REPORT NUMBER**
N/A

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Rome Air Development Center (COES)
Griffiss AFB NY 13441-5700

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
RADC-TR-90-404, Vol XI
(of 18)

**11. SUPPLEMENTARY NOTES** (See reverse)
RADC Project Engineer: John J. Crowter/COES/(315) 330-3564

This effort was funded partially by the Laboratory Director's fund.

**12a. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; distribution unlimited.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** (Maximum 200 words)

The Northeast Artificial Intelligence Consortium (NAIC) was created by the Air Force Systems Command, Rome Air Development Center, and the Office of Scientific Research. Its purpose was to conduct pertinent research in artificial intelligence and to perform activities ancillary to this research. This report describes progress during the existence of the NAIC on the technical research tasks undertaken at the member universities. The topics covered in general are: versatile expert system for equipment maintenance, distributed AI for communications system control, automatic photointerpretation, time-oriented problem solving, speech understanding systems, knowledge base maintenance, hardware architectures for very large systems, knowledge-based reasoning and planning, and a knowledge acquisition, assistance, and explanation system.

The specific topic for this volume is the use of logic Programming methodologies for knowledge base maintenance.

| 14. SUBJECT TERMS | 15. NUMBER OF PAGES |
|---|---|
| Artificial Intelligence, Prolog, Knowledge Base, Logic Programming, Languages, Feasibility, Knowledge Base Maintenance | 96 |
| | 16. PRICE CODE |

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

NSN 7540-01-280-5500
Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

Block 5 (Cont'd) Funding Numbers

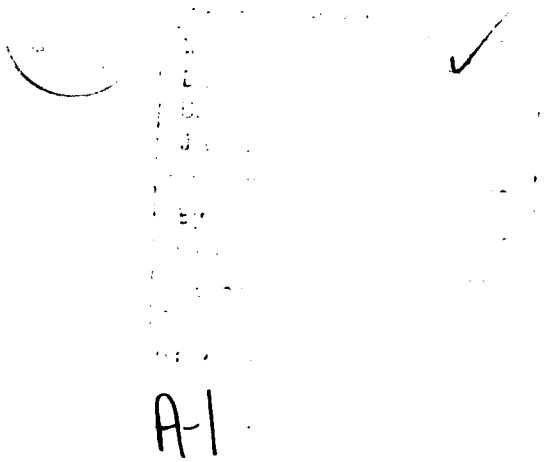
| | | | | |
|---|---|---|---|---|
| PE - 62702F | PE - 61102F | PE - 61102F | PE - 33126F | PE - 61101F |
| PR - 5581 | PR - 2304 | PR - 2304 | PR - 2155 | PR - LDFP |
| TA - 27 | TA - J5 | TA - J5 | TA - 02 | TA - 27 |
| WU - 23 | WU - 01 | WU - 15 | WU - 10 | WU - 01 |

Block 11 (Cont'd)

This effort was performed by the Syracuse University, Office of Sponsored Programs.

# Table of Contents





## 11.1 Executive Summary

This project was concerned with the development of logic programming-based machinery for the management of large complex knowledge bases of a highly dynamic character, together with the development of mathematical foundations for such systems. Knowledge base management includes the maintenance of ordinary integrity constraints as well as sophisticated reason maintenance systems. The work was carried out from the point of view of certain *meta-level* extensions of Prolog, generically baptised *metaProlog*. The primary tasks of the project included the following.

- Continued development of the metaProlog system. The principal goals here are the construction of an efficient metaProlog compiler, development of sophisticated memory-management methods, the development of suitable interfaces to non-metaProlog external databases, and the study of co-routining and concurrency.
- Development of knowledge representation formalisms in metaProlog, including analogs of frames, semantic nets, blackboards, etc.
- Study of the expression of generic database management and knowledge base reason maintenance approaches in metaProlog, with special attention begin devoted to maintenance of static and dynamic integrity constraints, reason maintenance, and daemons.
- Construction of one or more experimental demonstration systems using the machinery developed.
- Exploration of semantic foundations both for classical logic programming as well as non-standard approaches showing potential for dealing with the theoretical problems which arise in knowledge base maintenance.

We developed considerable knowledge of the structure and uses of the metaProlog system, ranging from its theoretical underpinnings to its use for implementing such programming constructs as frames, semantic nets, and message-passing. We also developed considerable expertise and tools concerning the implementation of systems of the character of Prolog and metaProlog. We first applied this to the construction of a byte-code interpreter-based compiler for Edinburgh Prolog which achieved 10K LIPs running the standard benchmark on a VAX 780. At the time, this was the

fastest implementation of Prolog on the VAX 780. This was used to compile our first substantial simulator for metaProlog (written in Prolog), producing a system which enabled us to begin serious metaProlog-based experiments. We then began extensions of the abstract machine underlying the Prolog byte-code interpreter aimed at producing an abstract machine suitable for the compilation of metaProlog. We explored a number of alternatives which presented themselves, eventually consolidating most of the valuable ideas into one system. Two alternative approaches to one aspect of the system led to the development of two alternative versions of the metaProlog compiler system. Both versions implement the core metaProlog features:

- Theories (logic databases) as first-class program objects which can be the values of variables and be returned by procedures;
- Direct program access to the underlying proof predicate.

Both are incremental and interactive compilers which appear to be interpreters, but which generate (very quickly) byte-coded instructions for the underlying abstract metaProlog machine; these instructions are executed by an abstract machine interpreter coded in C. (Following the pattern for ordinary Prolog, extremely efficient native code compilers can be developed from this architecture.) Both systems had approximately the same efficiency as our earlier byte-coded Prolog compiler: approximately 8-10,000 LIPS on the native reverse benchmark, depending on cache interaction, on a VAX 780.

Both systems were later extended to incorporate the following:

- Complete garbage collection; This included collection of compiled program code which is stored on the system heap.
- Proofs of goals as first-class objects; Consequently, programs can reason about the proofs resulting from solutions of goals (e.g., for explanation generation or for sophisticated fault diagnosis).

Very early on, we completed the axiomatization of a medium-scale knowledge-base problem in Edinburgh Prolog. We used this experience to guide some of the investigations into the design of the metaProlog engine. Later, we converted it to run in the metaProlog system.

We also conducted a extensive study of the truth and reason maintenance literature, eventually focusing primarily on deKleer's Assumption-Based Reason Maintenance. Implementation of ABRM can be carried out using metaProlog. However, because of the logical character of deKleer's work, we are studied methods of abstracting its basic facilities and directly incorporating them in metaProlog as system facilities.

We also constructed an interface from our orginal Prolog compiler to the academic version of the INGRES DBMS, and experimented with it. Because of the monolithic character of INGRES, communication between the two systems was limited to string-based communication, and the results were initially somewhat disappointing. However, we later ported the interface to the commercial version of the INGRES DBMS, and achieved much better results. This reinforced our conviction that there must be as close as possible communication between the Prolog/metaProlog system and any external DBMS system with which it is linked.

We examined a number of semantic approaches to clarifying the foundations of metaProlog. The fundamental difficulties arise from the "amalgamated character" of the language, wherein the variables of the language must not only range over conceptually ordinary individuals, but also over the syntactic constructs of the language itself, noting that the language is untyped (like LISP and ordinary Prolog). Several directions explored included using "possible world" semantics and a semantics in which ordinary logical structures interpreting the language are extended to include abstract syntactic entities generated (rather like a word algebra) from the individuals of the interpretation.

The first approach, while intuitively appealing, does not seem to lead to useful tools. The second approach has promise, but does seem to entail considerable complexity. However, a third approach (which is definitely related to the second) suggested itself, and this seems to have even greater potential. In essence, this approach follows the so-called "substitutional interpretation" of logic, but instead of basing the work on the traditional two-valued truth values, utilizes collections of partial search spaces for proofs in the language as the set of truth values.

We devoted considerable effort to exploring theoretical approaches to default reasoning, inconsistency, stratified knowledge bases, non-standard logics, topological semantics, and multi-valued logic programming. These investigations were quite successful. The details are presented in the body of the report.

## 11.2 Introduction

### 11.2.1 Logic and Databases: The Need to Extend Prolog

Prolog has many attractive features as a programming tool for artificial intelligence and the management of knowledge bases. These include code that is easy to understand, programs that are easy to modify, and a clear relation between its logical and procedural semantics. Moreover, it has proved possible to create clear and efficient implementations. Nonetheless, it possesses several shortcomings. Chief among these is difficulty representing dynamic databases (databases which change in time) and an apparent restriction to backward chaining, backtracking, depth-first search. A major component of our work has been to develop and implement an extension to Prolog, called *metaProlog*, which preserves the virtues of Prolog while introducing powerful constructions to attack these problems. This work is a direct continuation of the investigation into meta-level programming in logic begun by Bowen and Kowalski [1982].

Many artificial intelligence applications demand facilities which amount to the ability to dynamically manipulate databases or knowledge bases. A database is most naturally represented in Prolog as a set of assertions and clauses. This exploits all the advantages of Prolog's inherent deductive machinery. However, the logical core of ordinary Prolog provides no conceptual basis for segmenting or modifying the database. Most implementations of Prolog have provided ad hoc extensions to the basic logic programming paradigm which allow for dynamic modification of the program database by the program itself. But since the database is the program, the use of these facilities introduces difficulties similar to those introduced by global variables and self-modifying code in conventional programming languages. The effect of these features on the virtues listed above is catastrophic. Programs become difficult to understand, reliable modification of the code is almost impossible, and the logical semantics is utterly destroyed. We know of no mathematical or philosophical definition of first-order proof where the collection of axioms is not fixed. We would suspect any such notion to be incoherent. We believe these difficulties can be overcome by the introduction of theories as first-class objects which can be dynamically created and passed as parameters. In standard Prolog, goals are invoked with respect to a single background theory. In metaProlog, goals must be proved in an explicitly identified theory. We regard this system as simply a first-order logical theory of axiom sets and proofs.

The means of indicating that a meta Prolog goal G should be solved in a particular theory T is an explicit call on the proof predicate *demo*. From a logical point of view, the proof predicate is really a relation between three objects: the *theory T*, the *goal G*, and the *proof P* which attests to the solvability of G in T. But logic programming is not only concerned with the static existence of proofs, but also the process of discovering them. That is, it is also concerned with the notion of search space and search strategies. Thus, for logic programming, the deep central relation is the one which holds between a theory T, a goal G, and the complex object consisting of a proof for G in T seen as a portion of a search space explored by a particular search strategy. Our investigations have led us to the conclusion that all of these entities must be treated as first-class objects (metaProlog terms) capable of being manipulated and passed as values of parameters.

### 11.2.2 Meta-Level Programming

It is important to make clear our notion of meta-level programming. Briefly, one distinguishes between the formal language being used to conduct some (unspecified) axiomatic investigation (the object language) and the language used to carry on any discussion about the object language (the metalanguage). For many purposes (including those of this paper), the metalanguage need only be powerful enough to discuss the combinatorial syntactic properties of the object language. The essential point is that the relations of the metalanguage are about the syntactic entities of the object language: the variables of the metalanguage range over various syntactic entities of the object language. In contrast, the variables of the object language either have no specified range (when it is viewed as a formally uninterpreted language) or (when the object language is treated as being interpreted) range over the members (possibly extremely mathematically complex) of some specified set.

Properly viewed, an ordinary Prolog interpreter is already a meta-level object. The object level consists of a fragment of ordinary first-order logic, a language and proof predicate. The latter describes which formulas of the language are consequences of sets of other formulas of the language. The meta-level of a theorem-prover is concerned with the manipulation of sets of object-level formulas in the search for a collection of formulas which witnesses the derivability of a given goal formula from a given set of axiom formulas. The prover proper is a meta-level object because its variables range over formulas (and other syntactic classes) of the object level language.

Thus a Prolog interpreter really defines a relationship between sets of formulas (the program database), goal formulas, and proofs, namely the relation that the proof witnesses the deducibility of the goal formula from the program database. (Note that the standard Prolog interpreters return a portion of the proof to the user, namely that part of the substitution applying to the variables occurring in the goal). As commonly implemented, pure Prolog interpreters incorporate the program database as a fixed part of the interpreter. Thus, from a meta-level point of view, a standard Prolog interpreter provided with a fixed program database defines a certain meta-level unary predicate applying to goal formulas. This meta-level unary predicate holds for just those goal formulas which are deducible from the program database by the interpreter. The fundamental operator of standard Prolog systems is thus a one-place operator (usually written call(...)) which invokes a search for a deduction of its argument from the implicit program database parameter. The heart of the proposal set forth by Bowen and Kowalski was to utilize a system implementing the full deducibily relation described above. Such a system would have metavariables which not only range over formulas and terms, but would also allow the metavariables to range over sets of formulas (called theories). The fundamental operator of such a system is a three-place operator, usually written demo(Theory,Goal,Proof), which invokes a search for a proof of the goal formula appearing as its second argument from the theory (or program) appearing as its first argument.

All metaProlog program databases are the values of metaProlog variables and are set up either by reading them in from files or by dynamically constructing them using system predicates. Besides the built-in predicate demo/3, the system predicates include:

- add_to(Theory, Axiom, NewTheory)
- drop_from(Theory, Axiom, NewTheory)

which build new theories from old ones by adding or deleting formulas. Thus for example, one might find the body of a clause containing calls of the form

(*) ..., add_to(T1, A, T2), demo(T2, D, P),...

where the theory which is the value of T1 has been constructed by the earlier calls. The effect of (*) would then be to construct a new theory T2 resulting from T1 by the addition of the formula A as a new axiom, and then the invokation of a search for a proof of the formula D from the theory T2. Since demo implements the proof

relation, such programs as (*) preserve the logical semantics of Prolog while providing for the dynamic construction of new databases from old.

The correctness and completeness of an implementation of demo are expressed by what were called reflection rules by Bowen and Kowalski:

- If demo(T, A, P), then A is derivable from T via proof P.
- if A is derivable from T via proof P, then demo(T, A, P).

### 11.2.3 The General Situation

The real power (meta_power) of this system lies not in the specific system facilities we have described, but in the programming methodology they introduce. The example in the preceeding section only beings to explore the possibilities of this system. Using this approach, in [Bowen and Kowalski 1982], [Bowen and Weinberg 1985], and [Bowen 1985] we have begun to logically characterize frames and default hierarchies, generalized networks of theories and semantic nets, and more general control strategies such as bottom-up or breadth-first search. There is no logical requirement that the only notion of proof in metaProlog be the Horn clause-oriented demo predicate we have introduced. We see no reason why other methods of proof cannot co-exist with demo. We envisage the situation in which another method of proof would be rapidly prototyped using explicit recursive calls on the present demo, and later integrated into the system at a low level.

By stepping up to the full meta_level point of view wherein all components of the system have become first-class objects, we have entered the realm of a logical construal of Theories, Goals, and SearchSpaces in which it is possible to axiomatically and programmatically characterize elements of the system previously regarded as parts of the implementation. This allows us to introduce powerful logical approaches to the construction of artificial intelligence systems, and in particular, to systems which must manipulate complex knowledge bases.

### 11.2.4 Non-Classical Logic Programming

Classical logic is a logic of truth. Using classical logic we can reason about the truth of different kinds of propositions relative to a given theory. Thus, we classify propositions as either being *false* or *true*.

Unfortunately, this often proves to be an overly simplistic point of view. For instance, the famous *Fermat's Last Conjecture* must assuredly be either true or false, but at this point in time, we are unable to say, with certainty, which of these cases is the correct one. The same is also true of the $P = NP$ ? problem. There are those, however, who strongly disbelieve the proposition $P = NP$. Classical logic does not permit us to express this *disbelief* because, of course, these disbelievers may well turn out to be wrong, and it may indeed be proven a few years hence that $P$ is indeed equal to $NP$.

We view many non-classical logics as logics of *belief*. Typically, human beings are fallible. They have beliefs and disbeliefs, which may be wrong when judged relative to some empirical standard. The process of changing our beliefs is a common occurrence in our daily world. Often our beliefs turn out to be correct, but often they are not. Worse still, people may hold beliefs that are inconsistent in some respects – yet they may be able to reason perfectly well about certain other domains.

Our point here is simply that the study of beliefs is important, and perhaps even more important than the study of truth, which is after all a rather ephemeral quantity.

The approach of the metaProlog system is to make the various sets of beliefs (theories) explicit and amenable to direct manipulation. As such, the system provides the underlying machinery for several possible approaches to the problem of belief, but is in itself, neutral. The study of belief and the process of change of belief has been extensive. We devoted our major efforts in the following areas:

- A topologically motivated semantics for logic programs which has developed deep and powerful theorems concerning non-monotonic reasoning.
- A theory of logic programming that allows us to reason in the presence of inconsistency. In particular we develop one such logic which belongs to a family of logics that go by the generic name *paraconsistent logics*.
- A theory of logic programming that allows us to reason in the presence of uncertain information, i.e. information which is vague in the sense that one is

not sure of its truth/falsity, but has some feel (usually expressed in terms of a quantitative "certainty" factor) of the truth/falsity of a proposition.

- A family of of logic programming languages over multivalued logics having a certain kind of algebraic structure (i.e. a complete lattice). Under such circumstances, both the declarative (i.e. model theoretic and fixed point theoretic) semantics and the proof-theoretic (i.e. query processing procedures) generalize to the multivalued case.

### 11.2.5 Theoretical Basis for Logic Programming

As logic programming is a comparatively new field, we find that its basic theoretical underpinnings are very weak. There are many techniques in mainstream mathematics which may be used as *tools* to study the semantics of logic programming. It can hardly be doubted that establishing important links between well understood mathematical techniques and the semantics of classical and/or non-classical logic programming can only help enrich the semantics of logic programming. With this goal in mind, our group has undertaken the study of the topological and algebraic foundations of logic programming.

Associated with any (classical) logic program is an operator whose fixed-points are exactly the models of a formula called the completion of the program. One of the open problems in logic programming is to determine conditions for the completion to be consistent. A. Batarekh and V.S. Subrahmanian defined a (compact and Hausdorff) topology called the query topology on the space of interpretations of the first order language associated with a program. It is shown that whenever the program is either covered and/or function free, the operator associated with the program has a fixed-point iff it possesses a collapsibility condition. This collapsibility condition therefore yields a necessary and sufficient condition for program completions to be consistent (for such programs).

One can now study the algebraic properties of the space of programs by looking at the set of all operators associated with programs and associating some binary operators. Under some natural binary operators originally defined by Mancarella and Pedreschi, we obtain an algebra on programs that is easily seen to be a distributive lattice. The important question now is that of negation. Is there some notion of complementation relative to programs ? Unfortunately, there is no such notion of

complementation that yields a Boolean algebra or for that matter any richer algebraic structure like a ring, etc. (except in the most trivial cases). Finally, we can use this framework to study the equivalences of programs – in particular, a notion called subsumption equivalence due to Maher can be generalized considerably to normal logic programs and also to paraconsistent and/or multivalued logic programs.

## 11.3 Hamid Bacha

*metaProlog Implementation and Application*

### 11.3.1 Implementation of the metaProlog Compiler

The two major accomplishments of my work on the project are the completion of the metaProlog system and the implementation of a medical expert system in metaProlog. The metaProlog language is an extension of the popular logic programming language Prolog. As a high level programming language, Prolog has the most efficient implementation while still closely approximating the ideals of logic programming. Nevertheless, it has many limitations in terms of expressive power and problems with its ad hoc extra-logical features. These shortcomings have been recognized for a long time by many researchers, and a meta-level approach has been advocated as an alternative. Among the shortcomings that hamper the expressive power of Prolog are the many aspects that are supported by its underlying architecture, but not directly available to the user. Some of these aspects are:

- The sets of clauses (database)
- The provability relation ($\vdash$)
- The control strategy (depth first search, clause selection according to textual order)
- The rules of inference
- The proof trees

Some of the add hoc extra-logical features of Prolog that tend to cause problems are the "assert" and "retract" primitives which dynamically modify the database. The metaProlog system tries to deal with some of these shortcomings while preserving the ideals of the logic programming paradigm. The tacit and otherwise inaccessible aspects of the system it makes explicit include the provability relation (referred to as "demo"), the sets of relations or procedures (referred to as "theories"), the sets of clauses making up a procedure (referred to as a "viewpoint"), and the proof trees. The primitives "assert" and "retract" are replaced by "addto" and "dropfrom" which are used to create new theories from existing ones. Some definitions were introduced to extend the accepted Prolog terminology to cope with the use of multiple databases

(actually, instead of saying we use multiple databases, we prefer to say we have one database which contains multiple theories). These definitions are:

- A metaProlog database is a collection of theories and relations (procedures).
- A relation is a collection of beliefs.
- A theory is a collection of viewpoints.
- A viewpoint is a set of related beliefs (equivalently, a subset of the set of beliefs making up a relation).
- A belief is a metaProlog fact or rule.

As we can see from these definitions, the metaProlog database contains theories, and the theories contain viewpoints. A built-in inheritance mechanism lets theories share clauses, thus avoiding the prohibitive cost of copying clauses from theory to theory. A fast algorithm is used to match the theories with their corresponding viewpoints.

Proofs, in the form of proof trees, are directly available to the metaProlog user. They are treated as first-class objects and can be manipulated very much like any other metaProlog terms. Since they include all the subgoals that participate in the evaluation of a given goal, they can be used, for example, to generate explanations for applications involving expert systems. An unexpected but pleasantly surprising use of proof trees is to affect the control strategy of the system by directing the search for a solution along a more desirable path. Indeed, if the system is presented with a goal and a proof tree indicating a possible solution, it only needs to check whether there is a proof for the stated goal along the branches of the search space corresponding to the given proof tree. In other words, the proof tree guides the search for the solution. No other possibly wrong or infinite paths need to be followed during the evaluation of the goal. No other solution needs to be considered. We can also use proof trees that are only partially instantiated. That is, a skeletal description of some desirable features we would like to see participate in the solution. In this case, the partially instantiated proof tree serves to focus the system's attention on specific portions of the search space, leaving it free to explore within these selected subspaces. Early pruning of non-fruitful branches of the search space and avoidance of blind alleys may lead to a more efficient solution for certain types of problems, despite the overhead associated with the proof trees.

The extensions mentioned above were achieved in the context of a compiled approach based on the Warren Abstract Machine architecture. This resulted in a fast and efficient system which relies on an interactive incremental compiler for flexibility and ease of use. The objective of these extensions is to provide a richer and more expressive language, as well as a more accommodating environment for artificial intelligence applications such as knowledge representation, natural language processing, and expert systems.

### 11.3.2 Application of metaProlog to Medical Expertise

To test the suitability of metaProlog for large scale applications, we embarked on the task of implementing a medical expert system. The area of expertise selected was that of Acid-Base and Electrolyte Disorders. The goal was to integrate the clinical knowledge with the pathophysiological knowledge to come up with a robust expert system that combines both surface-level and deep-level reasoning. The system built used some innovative features such as:

- First-principles assisted evidential reasoning: This method relies on the more prevalent and widely used clinical knowledge for diagnostic purposes, but brings in the pathophysiological knowledge on an as needed basis.
- Progressively expanding diagnostic possibilities: Meta-level knowledge and priorities are used to restrict the search for the diagnosis to the more promising leads. These restrictions are then progressively lifted to include more and more possibilities for consideration. This method provides a more focussed approach and a better interaction between the system and the user.
- Thesaurus-driven user interface: all the interactions between the system and the user are carried through the user interface. To enhance the friendliness of the system, the user interface is coupled with a thesaurus that defines all the terms of interest in the domain of the expert system. The thesaurus specifies the type of query to be used with each term and the type of answer to expect. It lists the variations as well as the qualifiers applicable to each term. Whenever possible, it specifies the precondition that must hold before the user can be queried about a certain finding.

The preliminary results from our experiments with this system were very promising and seem to suggest that we have an adequate approach. More important, the

whole experience in implementing this system points to the usefulness and suitability of metaProlog for implementing expert systems. The metaProlog system offers both a functional design advantage in terms of knowledge representation and hypotheses exploration, and a software engineering advantage in terms of structuring the expert system shell.

### 11.3.3 Future Work

Unlike many researchers who rely mainly on meta-interpreters to obtain the advantages of the meta-level approach, we went one step further and showed that it is possible to have some of these same advantages plus the speed of a compiler. However, only some of the desired features of metaProlog have been implemented in this first phase. The next phase should address the following points:

- Explicit control: we should be able to specify the control regime to used to solve any goal or subgoal. We should be able to choose between depth-first, incremental iterative deepening, or breadth first strategies. We should also have some way of specifying the order of the clauses when there are many alternatives.
- A choice of forward or backward chaining. This issue is tied to the control strategy above.
- A delay mechanism for waiting for some variables to be bound to ground terms.
- A mechanism for allowing coroutining to take place.
- Incomplete theories, that is theories that are not completely specified
- Explicit quantification

In addition to the direct work on the metaProlog system, suitable projects in various areas of Artificial Intelligence should be identified and implemented in metaProlog. These projects should be large and realistic enough to test the limits of the system. The lessons to be learned from these projects should hopefully confirm the viability of the many features of metaProlog and help establish it as a major player in the area of research and development of Artificial Intelligence systems.

### 11.3.4 Publications

*Meta-level Programming: A Compiled Approach.* **Proceedings of the Fourth International Conference on Logic Programming.** Melbourne, Australia, 1987. Edited by Jean-Louis Lassez.

*MetaProlog Design and Implementation.* **Proceedings of the Fifth International Conference on Logic Programming.** Seattle, Wa. 1988. Edited by K.A. Bowen and R. Kowalski.

*Beyond the WAM: A PAM for the CAM. (A Prolog Abstract Machine for Content-Addressable Memory.)* Submitted to the **6th International Conference on Logic Programming** to be held in Lisbon, Portugal in June 1989.

*Program Verification Using Meta-Level Logic Programming.* Submitted to the **6th International Conference on Logic Programming** to be held in Lisbon, Portugal in June 1989. (In collaboration with Sanjay Khanna).

*Clinical vs Pathophysiological Knowledge in Medical Expert Systems.* In preparation (in collaboration with Dr K.A. Bowen and Dr C. Carvounis). To be submitted to a medical journal.

## 11.4 Aida Batarekh

*Topological Aspects of Logic Programming*

### 11.4.1 Query Topology

Part I of this report was done in collaboration with V.S. Subrahmanian. A topology on the set of interpretations of a logic program $P$ was defined, and its properties studied. The Query topology is defined as follows: the open sets are the collection of all subsets of $X$ which satisfy a (possibly infinite) disjunction of individual existential queries. If all $L_i$'s are positive literals, the query is said to be positive, if all $L_i$'s are negative literals, the query is negative. We show that the Query topology gives rise to a totally disconnected, Tychonoff, complete and metrizable space.

A study of equivalences of sentences based on classical and non-classical logics was also pursued. We proposed four notions of equivalences of sentences. We showed that under certain conditions on the lattice structure of the set of truth values of the logic of interest, three of these notions can be captured in terms of results on the convergence of monotone nets in topology, while the fourth notion can be captured in terms of a property of convergent nets in compact Hausdorff spaces which is what the Query topology gives rise to. Our work may be viewed as a semantical counterpart of Maher's [6] syntactical characterization of pure 2-valued logic programs. These results [1] can be found in the technical report "Semantical Equivalences of (Non-Classical) Logic Programs", which has also been presented at the 5th International Conference on Logic Programming, August 88, Seattle.

Further investigations into the notion of axiomatizability, which we have previously defined, lead to the study of the special case of *finitely definite clause axiomatizability* or FDC-axiomatizability for short. We study mappings which are FDC-deformations, i.e., mappings from sets of interpretations into sets of interpretations such that the property of FDC-axiomatizability is preserved. We narrow the Query topology to the set of FDC-axiomatizable interpretations and investigate whether special properties can be obtained.

### 11.4.2 Topological Approaches to Non-Monotonic Reasoning

I studied the connection between the Query topology and the well-known Scott topology [2] and established the following: the collection of all open sets which satisfy a disjunction of *positive* queries, are exactly the open sets in the Scott topology. Similarly, the collection of all open sets which satify a disjunction of negative queries are the open sets in the Inverse Scott topology, which I have defined in a manner symmetrical to that of Scott topology. It is also shown that the Inverse Scott topology is distinct from the dual of the Scott topology. These results and some properties of the Query topology can be found in the technical report [2] referenced below.

The lattice of interpretations is shown to be algebraic and supercontinuous hence also complete and continuous. Therefore one can compare the Query topology to the Lawson topology which is defined only on continuous lattices. The relationship between the two is established: the open sets in the Query topology are exactly the open sets in the Lawson topology. This in turn was used to prove that the space was compact and 0-dimensional, hence that it had a countable base of sets which are both open and closed. These results appear can be found in [3].

A notion of *axiomatizability* is introduced and a set $Y$ of interpretations is shown to be axiomatizable with a set $S$ of clauses if and only if $Y$ is closed in the Query topology. These results and others pertaining to the applications of topology in Logic Programming have been collected in a technical report [4] which will be submitted to a journal for possible publication.

Having proved that the space of interpretations under the Query topology is a complete metric space, i.e. that there exists a metric which metrizes the space, the nature of the metric is investigated. I show that the Query topology is a Cantor space, and that there is a homeomorphism between the space of interpretations under the Query topology and the Cantor set.

The potential applications of the results found for the Query topology are studied with respect to non-monotonic deduction operators occuring in the underlying language. Specifically, an attempt is made at modifying the Query topology to deal with the non-monotonic operator $\nabla$ introduced in my dissertation and used to introduce assumptions.

An investigation into the continuity properties (continuity as defined in topology) of the well known deduction operators $T \uparrow \alpha$ and $T \Uparrow \alpha$ shows that they are continuous over the set of Herbrand interpretations of a pure logic program $P$ provided $P$ has

no clauses with free variables and $\alpha < \omega$.

### 11.4.3 Publications

[1] Batarekh A. and Subrahmanian V.S., (1988) *Semantical Equivalences of (Non-Classical) Logic Programs*, 5th International Conference on Logic Programming, August 88, Seattle.

[2] Batarekh A. and Subrahmanian V.S., (1988), *The Query Topology in Logic Programming*, Proc. Intl. Symp. on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, Springer Verlag, Feb. 1989.

[3] Batarekh A. and Subrahmanian V.S. (1988), *Topological Model Set Deformations in Logic Programming*, to appear in: Fundamenta Informatica.

[4] Batarekh A. and Subrahmanian V.S. (1988), *A T-4 Space of Models of Logic Programs and their completions, I: Foundations*, Technical Report, Logic Programming Research Group, LPRG-TR-88-15.

### 11.4.4 References

[5] Gierz G., Hofmann K.H., Keimel K., Lawson J.D, Mislove M. and Scott D.S, (1980), *A Compendium of Continuous Lattices*, Springer-Verlag.

[6] Maher M. (1986) *Equivalences of Logic Programs, in: Foundations of Deductive Databases and Logic Programming*, (ed. Jack Minker), Morgan Kaufmann.

## 11.5 Howard A. Blair

### *Theory of Logic Programming*

Howard Blair and Krzyzstof Apt of the University of Texas at Austin and CWI Amsterdam, the Netherlands, studied the recursion-theoretic complexity of the perfect (Herbrand) models of stratified logic programs. This work culminated in [AB88]. They showed that these models lie arbitrarily high in the arithmetic hierarchy. As a byproduct they obtained a similar characterization of the recursion-theoretic complexity of the set of consequences in a number of formalisms for nonmonotonic reasoning. They showed that under some circumstances this complexity can be brought down to recursive enumerability. This work continued the investigation into the properties, both semantic and proof-theoretic, of stratified programs introduced by K. R. Apt, H. A. Blair, A. Walker and A. Van Gelder.

This earlier work is embodied in [ABW87,VG87]. An earlier version of [ABW88] was issued in 1986 as an IBM Thomas J. Watson Research Center (Yorktown Heights) technical report. This was seminal work that introduced the theory of stratified logic programs. The book containing the paper was at last published, nine months behind schedule, in March 1988. The results reported in the "Arithmetic Classification" paper and related results were discussed but not proved in early drafts of [ABW87]. ([VG87] also appeared in an earlier version in [VG86].) Revising and final editing of [ABW87] was supported by both the Project and a small grant from the Syracuse University Faculty Senate to support a graduate student from the People's Republic (sic) of China.

Following the Logic Programming conference in August, 1988, the "Arithmetic Classification" paper was invited for submission to the journal *Fundamenta Informatica*. It is now to appear, and amalgamates the results contained in [AB88a]. This latter technical report had been intended for seperate publication but the administrative demands of the Project preempted the time required to develop this publication: the delay forced the amalgamation of the results with those of the "Arithmetic Classification" paper. The Completions of Recursion-free general programs, together with a first-order domain closure assumption, constitute complete theories. The standard model of a recursion-free program is decidable.

These combined results were also presented by Blair at a colloquium talk in April, 1988 at the State University of New York at Albany.

Meanwhile, in 1986 Howard Blair continued his study of the recursion-theoretic complexity of the structure of Herbrand bases of programs and reported this in [Bl86]. These results show that a variety of properties of programs, viz. canonicality, determinateness, etc. are highly undecidable and exact hierarchical lower bounds are given.

In [Bl87] Blair showed that all recursive relations over a given finitely generated first-order language are computable by determinate programs; i.e. the *canonical* programs of Jaffar and Stuckey (1986) with completions that have exactly one Herbrand model. In [Bl87] it is further shown that all recursively enumerable sets over the Herbrand universe of a finitely generated language $L$ are computable by either *success* or *failure* sets of canonical programs. Indeed, for any two recursively enumerable disjoint subsets of the Herbrand universe of $L$, a single program can be found that has one of the sets as its success set, and the other as its finite-failure set. The results are effective in the sense that from indices of two r.e. sets, the required programs can be constructed.

Jaffar and Stuckey [JS86] define *canonical* logic programs and show that for each logic program there exists a semantically equivalent (formalized by a definition of conservative extension) logic program which is canonical. [Bl87] gives a different construction that strengthens the previous results in [JS86]. Given a logic program $P$, we effectively construct a program $P'$ such that with respect to the classical definition of *conservative extension* the following holds:

(i) $P'$ is a conservative extension of $P$ in the classical sense, and comp($P'$) is a conservative extension of comp($P$), in the sense of [JS86]
(ii) $P'$ is canonical; that is, $\mathbf{T}_{P'} \downarrow \omega = \mathrm{gfp}(\mathbf{T}_{P'})$.
(iii) $P'$ contains no new constant or function symbols, and as a consequence of this must necessarily contain new predicate symbols if $P$ itself is not canonical.
(iv) $P'$ has the same finite failure set, when restricted to the Herbrand base of $P$, as $P$.
(v) $P'$ has the same success set, when restricted to the Herbrand base of $P$, as $P$.

In [Bl88,Bl88a] *direct universal computability* by logic programs is defined. When the set of function symbols of $L$ is infinite there are recursive subsets of the Herbrand universe and Herbrand base of $L$ which are not computable by any logic program. The availability of an effective enumeration of the Herbrand universe $U_L$ of $L$ for inclusion in programs is shown to be *necessary* and sufficient for direct universal computability by logic programs with respect to $L$. We then show this result holds with respect to completions of normal programs as well.

During the academic year 1987-1988 an initial draft of [BBS87] was completed.

This paper presents the thesis that a logic program $P$ without negation, over a variant logic, is a theory that can be associated with an operator whose prefixed points are exactly the models of P. Part II of this paper to be entitled, "A Logic Programming Semantics Scheme, Part II: DOXOLOG, a Belief Maintenance Language", is concerned with an application of the semantic approach of part I to give a formal semantics for the language DOXOLOG, indicated in the above title.

During 1988 Blair continued his investigations of morphisms in logic programming model theory. This work is an attempt to, in particular, model-theoretically formalize the semantics of database updates. The idea is that a new database instance is a morphic (roughly homomorphic) image of a previous database instance such that both instances are models of the same theory of the database's integrity constraints.

During 1988 a draft of a proposal for research on Computational Reasoning with Nonclassical and Paraconsistent Logics was prepared. The draft is intended to serve as a master document from which various proposals can be derived by both Blair, and now independently, his student V. S. Subrahmanian, who has now completed his Ph.D., having graduated in August.

In February, 1988 a revised version of [BS87] was invited for submission to a special issue of the journal *Theoretical Computer Science* for a special issue on selected papers from the Seventh Conference on Foundations of Software Technology & Theoretical Computer Science. The paper was subsequently accepted. The idea of *paraconsistency* is that a collection of propositions may be locally consistent, but globally inconsistent. Thus one can reason with large globally inconsistent sets of

sentences while taking care to avoid inconsistent lines of reasoning as they are detected.

Some progress was made in calendar year 1988 on "An Inductive, Stratification-free Definition of Standard Models of Stratified Logic Programs". It is still in a formative stage. The work that needs to be done to establish this 'definition' requires that a limit of an alternating operator, recursive in zero-jump, exist in the right circumstances.

Two other results obtained during Fiscal year 1988 were that Blair developed a functionally oriented theory of nondeterministic partial recursive functions in accord with an earlier theory, relationally oriented, of such functions advanced by Ashok Chandra, and this theory was applied to showing that a logic program with a well-founded dependency relation forms a $\Pi^1_1$-complete set.

A theorem which constructively establishes a domain over which every logic program is canonical was conjectured, with the general outlines of how to prove it, recently in March, 1989 by Howard Blair and Allen L. Brown of both Xerox Webster Research Center's System Science Laboratory and the School of Computer and Information Science at Syracuse University. Subsequently it was discovered that only a weaker version of the theorem that was initially conjectured in March could be proved with the techniques that had been worked out by Blair and Brown in March, April, and May of 1989. During June of 1989, Blair and Brown discovered that the techniques *could* be coupled with an iterative technique and limit construction to prove the desired theorem. A draft of a paper describing the theorem and the techniques used to prove it was written in July, 1989. Below, we briefly describe the theorem and construction which comprises Blair's main effort during Fiscal Year, 1989. During the period from May 15 to August 15, 1989, Blair was supported *soley* by Xerox Corporation, Webster Research Laboratory on site — Blair was not supported during this period by the NAIC grant.

The model-theoretic semantics of completed definite clause logic programs exhibit an asymmetry regarding Herbrand models: the "duality between True and False, success and failure, least fixed point and greatest fixed point, least [Herbrand] model and greatest [Herbrand] model" (*cf.* [JLM86]) breaks down for various definite clause programs. As Jaffar, Lassez, and Maher [JLM86] point out, for those definite clause

programs $P$ for which $\mathbf{T}_P \downarrow \omega = \text{gfp}(\mathbf{T}_P)$, many aspects of the theory of such programs are symmetric. ("gfp" ["lfp"] means "greatest [least] fixed point".) Jaffar and Stuckey [JS86] call a definite clause program $P$ with the property that $\mathbf{T}_P \downarrow \omega = \text{gfp}(\mathbf{T}_P)$ a *canonical* program and show that every definite clause program $P$ has, in a suitable sense, a *conservative extension* to a program $P'$ that is canonical. (Hereafter we shall frequently use the term *program* synonomously with the term *definite clause program*.) Via such extensions, "the class of canonical programs is representative of the class of all programs." [JLM86].

An unfortunate choice of definition in [JS86] for the class of partial recursive functions (in which the minimization operator is applied only to total recursive functions) appears at first glance to lead to an ineffective construction of the canonical conservative extension. The Jaffar-Stuckey construction adds new constant and function symbols as well as new predicate symbols, in forming the extension. Blair [Bl87] reformulates the Jaffar-Stuckey construction in order to avoid expanding the given program's Herbrand universe and observes that the Jaffar-Stuckey construction is actually effective since, given a program $P$, an index for the finite-failure set of $P$, which is recursively enumerable, can be effectively obtained. From that index together with an application of the Normal Form Theorem, an explicit definition of a partial recursive function that enumerates the finite failure set of $P$ can then be obtained in which minimization is applied, once only, to an explicit definition of a primitive recursive function. In passing, [Bl87] also observes that if $R$ is a recursive subset of the Herbrand universe of a language $\mathcal{L}$ with only finitely many constant and function symbols, then the construction yields a *determinate* program $Q$ that computes $R$. Specifically,

$$\mathbf{T}_Q \uparrow \omega = \text{lfp}(\mathbf{T}_Q) = \text{gfp}(\mathbf{T}_Q) = \mathbf{T}_Q \downarrow \omega \ .$$

The previous review points out that to obtain a symmetric theory of programs in the sense of [JLM86] one can move in the direction of restricting the class of programs considered to those which are still representative, in a suitable sense, of all programs. Alternatively, one could move in the direction of relativising the $\mathbf{T}$ operators to a pre-interpretation $\mathcal{C}$. (Below, we will use the term *prestructure* instead of *pre-interpretation* since we feel that it is more in accord with usage in the literature of mathematical logic. Actualy, a prestructure for $\mathcal{L}$ is an *algebra* whose signature is $\mathcal{L}_=$.) If we undertake the latter move then we require a prestructure $\mathcal{C}$ (for language $\mathcal{L}$) such that it satisfies the Clark equality axioms [*cf.* [Ll87], [Cl78]] and for every program $P$ over language $\mathcal{L}$

$$\mathbf{T}_P^{\mathcal{C}} \downarrow \omega = \text{gfp}(\mathbf{T}_{\mathcal{C}})P \ .$$

$\mathcal{L}$ can be large, i.e., it can have *infinitely many function and predicate symbols*. Thus up to renaming of symbols, *every* program is canonical with respect to $\mathcal{C}$.

The construction proceeds in the following way: We begin with a first-order language $\mathcal{L}$ and a prestructure $\mathcal{W}_0$ for $\mathcal{L}$ that satisfies the Clark equality theory with language $\mathcal{L}$. Next, we define the notion of BF-trees with respect to $\mathcal{W}_0$. BF-trees, *simplicitur*, are defined in [WML84]. BF-trees are at the opposite end of a spectrum of generalizations from SLD-trees. In [WML84] both SLD- and BF-trees are special cases of objects called GLD-trees. In SLD-derivations an atom is selected from a goal and the goal is resolved with a clause in the given program using the selected atom and the head of the clause. In GLD-derivations a sublist of atoms occurring in a given goal is selected for resolution using the heads of a list of clauses from the program. In a BF-derivation the entire goal is "selected." GLD-, SLD- and BF-*trees* depict the disjunctive alternatives in sequences of GLD-, SLD, and BF-derivations, respectively.

A node in a BF-tree is, in general, labeled with a goal, a list of clauses, and a most generally unifying substitution. In a BF-tree *with respect to* $\mathcal{W}$ nodes are labeled with mgu's of equivalences of partially interpreted terms, i.e., terms having some of their variables assigned to individuals of $\mathcal{W}$.

We use the BF-trees *with respect to* $\mathcal{W}_0$ that we shall define to obtain a class of substitutions which yields a quotient of the set of terms of $\mathcal{L}$ that are partially interpreted in $\mathcal{W}_0$. The quotient, $\mathcal{W}_1$, also satisfies the Clark equality theory. It is obtained in a manner which resembles the prestructure obtained in the *proof* of the completeness of negation as failure due to Wolfram, Maher, and Lassez, [WML84].

From $\mathcal{W}_n$ we obtain $\mathcal{W}_{n+1}$ in the same manner as $\mathcal{W}_1$ is obtained from $\mathcal{W}_0$. If one carefully manages the selection of variables of $\mathcal{L}$ in constructing each $\mathcal{W}_{n+1}$ from $\mathcal{W}_n$ it turns out that the sequence of prestructures

$$\mathcal{W}_0,\ \mathcal{W}_1,\ \cdots\ \mathcal{W}_i,\ \cdots$$

forms a chain where each $\mathcal{W}_n$ is embedded in $\mathcal{W}_{n+1}$ and each member of the chain satisfies the Clark equality axioms. If for each $n$ we identify the individuals of $\mathcal{W}_n$ with their images in $\mathcal{W}_{n+1}$ then we have the union $\mathcal{W}_\omega = \bigcup_{n=0}^{\infty} \mathcal{W}_n$ is well-defined as a prestructure and satisfies the Clark equality theory. And $\mathcal{W}_\omega$ has the property that all programs over $\mathcal{L}$ are canonical with respect to it. In particular, $\mathcal{W}_0$ can be chosen to be the Herbrand universe of $\mathcal{L}$.

A revision of the initial draft of this paper is in preperation (September, 1989).

Blair has been invited to lecture on this work at both the Cornell University Mathematics Center and the upcoming *International Symposium on Mathematics and Artificial Intelligence* in January, 1990.

# References


[ABS8] Apt, K. R. & Blair, H. A. "Arithmetic Classification of Perfect Models of Stratified Programs". Invited Submission to *Fundamenta Informatica* (To appear.)

[AB88a] Apt, K. R. & Blair, H. A. *Recursion-free Logic Programs.* Logic Programming Research Laboratory Technical Report LPRG-TR-88-12

[ABW87] Apt, K., Blair, H., & Walker, A. "Towards a Theory of Declarative Knowledge", in *Foundations of Deductive Databases and Logic Programming,* Jack Minker, ed. Morgan-Kaufmann, Los Altos, CA., 1987, pp. 89-148.

[BBS87] Blair, H. A., Brown, A. L. and Subrahmanian, V. S. *A Logic Programming Semantics Scheme, Part I.* Jan, 1988. Syracuse University Logic Programming Research Group Technical Report LPRG-TR88-8.

[Bl86] Blair, H. A. "Decidability in the Herbrand Base". Workshop on Deductive Databases and Logic Programming, Washington D.C. Aug 18-22, 1986. Revised as Syracuse University Logic Programming Research Group Technical Report LPRG-TR88-13.

[Bl87] Blair, H. A. "Canonical Conservative Extensions of Logic Program Completions". *IEEE Symposium on Logic Programming*, San Francisco, August, 1987. pp. 154-161. Syracuse University Logic Programming Research Group Technical Report LPRG-TR88-14.

[Bl88] Blair, H. A. "Metalogic Programming and Direct Universal Computability". Revised Version to appear in *Proceedings of the Meta88 Workshop*, H. Abramson & M.H. Rogers, (eds.), MIT Press.

[Bl88a] Blair, H. A. "Metalogic Programming and Direct Universal Computability". Syracuse University Logic Programming Research Group Technical

Report LPRG-TR88-23. Appears in *Proceedings of Meta88: Workshop on Meta-programming in Logic Programming.*

[BS87] Blair, H. A. & Subrahmanian, V. S. "Paraconsistent Logic Programming" (Preliminary Version) Seventh Conference on Foundations of Software Technology & Theoretical Computer Science. December, 1987. pp. 340-360. Invited for Submission to *Theoretical Computer Science.* (To appear.)

[Cl78] Clark, K. L. "Negation as Failure." *Logic and Databases*, Gallaire, H., and Minker, J. (eds.), pp. 293–324, 1978.

[JLM86] Jaffar, J., Lasssez, J-L. and Maher, M. J. "Some Issues and Trends in the Semantics of Logic Programming," *Proceedings of the Third International Conference on Logic Programming*, Ehud, Shapiro (eds.). London, July 1986, pp. 223–241. *Lecture Notes in Computer Science*, no. 225. Springer-Verlag, 1986.

[JS86] Jaffar, J. and Stuckey, P. J. "Canonical Logic Programs". *Journal of Logic Programming,* vol. 3, no. 2 pp. 143-155, 1986.

[Ll87] Lloyd, J. W. *Foundations of Logic Programming*, (2nd. edition.) Springer-Verlag, 1987.

[VG86] Van Gelder, A., "Negation as Failure Using Tight Derivations for General Logic Programs" in: *Proc. of the 3rd IEEE Symposium on Logic Programming*, Salt Lake City, Utah, 1986.

[VG87] Van Gelder, A., "Negation as Failure Using Tight Derivations for General Logic Programs" in *Foundations of Deductive Databases and Logic Programming,* Jack Minker, ed. Morgan-Kaufmann, Los Altos, CA., 1987, pp. 149-176.

[WMLS4] Wolfram, D.A., Maher, M.J. & Lassez, J-L. "A Unified Treatment of Resolution Strategies for Logic Programs," *Proceedings of the Second International Logic Programming Conference*, Uppsala, Sweden, pp. 263-276, 1984.

## 11.6 Kenneth A. Bowen:

*Foundations and Application: Reason Maintenance*

### 11.6.1 The Logic of Monotonic Reasoning

**Introduction** An intelligent artifact (program) reasoning about some aspect of the world must often maintain a collection assertions representing its current beliefs. In many settings, these assertions may only be plausible conjectures which may have to be retracted depending upon the course of the agent's reasoning and the accumulation of evidence. in such circumstances, the agent's reasoning is said to be *non-monotonic*. If the assertions added to the collection of beliefs are never retracted, the agent's reasoning is said to be *montonic* [refs].

The course of development of the set of beliefs is dependent both on the (external) evidence discovered and on the agent's choices of resoning steps to employ, even in the montonic case. This picture of monotonic reasoning is strikingly similar to the intuitionistic descriptions of the idealized mathematician [ref-Brouwer] and its formalization in the *theory of constructions* [refs]. While the treatment of infinite totalities in intuitionism is a fascinating and problematic topic, a stric point of view can maintain that such totalities are only *potential* and never *actual*. This position is founded on the view that the idealized mathematician is a finite being acting in time, and that all actualized mathematical entities must be constructed by the mathematician. Consequently, not only must all actual entities be in fact finite, but the totality of constructed entities and verified assertions is necessarily finite at any point in time.

This strict finiteness of the collection of entities and verified assertions is certainly characteristic of the belief sets of intelligent computer programs. But like the intuitionist mathematician's sets, these sets are *potentially infinite* in that there is no in principle bound on the effort of either the ideal mathematician or the intelligent program.

Kripke's introduction of a classical model theory of intuitionistic logic provided a powerful tool for the classical analysis of intuitionistic reasoning. This model theory is also very attractive for the analysis of the reasoning of intelligent artifacts (in the non-monotonic case, in its *modal* incarnation). While the introduction of Kripke models provided a intuitively appealing set-theoretic interpretation of intuitionistic

$$\rightarrow -IS: \frac{\mathbf{A}, \Pi \Rightarrow \mathbf{B}}{\Pi \Rightarrow \Sigma, \mathbf{A} \rightarrow \mathbf{B}} \qquad \neg -IS: \frac{\mathbf{A}, \Pi \Rightarrow}{\Pi \Rightarrow \Sigma, \neg \mathbf{A}}$$

$$\wedge -IS: \frac{\Pi \Rightarrow \mathbf{A} \quad \Pi \Rightarrow \mathbf{B}}{\Pi \Rightarrow \Sigma, \mathbf{A} \wedge \mathbf{B}} \qquad \vee -IS: \frac{\Pi \Rightarrow \mathbf{A}}{\Pi \Rightarrow \Sigma, \mathbf{A} \vee \mathbf{B}} \qquad \frac{\Pi \Rightarrow \mathbf{B}}{\Pi \Rightarrow \Sigma, \mathbf{A} \vee \mathbf{B}}$$

$$\forall -IS: \frac{\Pi \Rightarrow \mathbf{A}}{\Pi \Rightarrow \Sigma, \forall \mathbf{x} \mathbf{A}} \qquad \exists -IS: \frac{\Pi \Rightarrow \mathbf{A}_{\mathbf{x}}[bfa]}{\Pi \Rightarrow \Sigma, \exists \mathbf{x} \mathbf{A}}$$

provided the eigen-variable condition is met

where we use the substitution notation of Shoenfield[4].

Figure 1: IS-Rules for LJ'

statements, these models have the disadvantage that in dealing with arithmetic and analysis, or for that matter, any theory in which all of the statements "there exist of least n individuals" are derivable, the interpretation requires that infinitely many individuals actually exist at each world-point or situation, thus preventing a direct interpretation of any concepts of potentially infinite totality. This drawback also applies to the analysis of the reasoning of artificial agents. In this paper, we provide a modification of Kripke's approach which allows us to restrict the number of individuals actually existing at any world-point or siutation to be finite. The price we pay is that the number of world-points is necessarily infinite, and in the interpretation of the logical operators, we must universally quantify over subcollections of the universe of situations. Consequently, as an analysis of intuitionistic reasoning, it remains thoroughly classical. However, as an analysis of the reasoning of intelligent agents, it provides an initial framework for the global analysis of the agent's reasoning.

**I-Structures and Validity** We will consider languages L with the logical symbols $\neg, \wedge, \vee, \rightarrow, \forall, \exists$, and $=$, together n-ary predicate symbols $\mathbf{p}, \ldots$, and function symbols $\mathbf{f}, \ldots$ for various $n \geq 0$. The system LJA is as defined in Gentzen [2]. The system LJ' is defined as follows (cf. Prawitz [3]). Sequents $\Gamma \Rightarrow \Delta$ are permitted to have more than one formula in the succedent $\Delta$. The axioms, structural rules, and the rules $\rightarrow -IA$, $\wedge -IA$, $\vee -IA$, $\neg -IA$, $\forall -IA$, and $\exists -IA$, are just as for LK or LJ. The rules for introduction in the succedent are as shown in Figure 1.

If $\Gamma'(\Delta')$ is a permutation of $\Gamma(\Delta), \Gamma' \Rightarrow \Delta'$ is a *variant* of $\Gamma \Rightarrow \Delta$. Obviously any

sequent provable in LJ is provable in LJ', and it is easy to prove the following lemma by induction on the complexity of proofs (cf. [3]).

**Lemma 1** If $\Gamma \Rightarrow \Delta$ is provable in LJ', either $\Gamma \Rightarrow$ is provable in LJ or for some $\mathbf{A} \in \Delta$, $\Gamma \Rightarrow A$ is provable in LJ.

**Corollary 1** A sequent $\Gamma \Rightarrow \mathbf{A}$ is provable in LJ if and only if it is provable in LJ'.

**Def 2** If $a_1, a_2, \ldots,$ and $b_1, b_2, \ldots,$ are all terms of L, we will call formulas of the form $\mathrm{a} = \mathrm{a}$ *identity axioms* and we will call formulas of either of the two forms

$$\mathrm{a}_1 = b_1 \wedge \ldots \wedge a_n = b_n \rightarrow f(a_1, \ldots, a_n) = f(b_1, \ldots, b_n)$$

$$\mathrm{a}_1 = b_1 \wedge \ldots \wedge a_n = b_n \rightarrow (p(a_1, \ldots, a_n) \rightarrow p(b_1, \ldots, b_n))$$

*equality axioms*, where in the latter, $p(a_1, \ldots, a_n)$ could be $\mathrm{a}_1 = \mathrm{a}_2$.

**Def 3** We will say that a sequent $\Gamma \Rightarrow \Delta$ is *provable* in LJ= if there exists a finite set $\Pi$ of universal closures of equality and identity axioms such that $\Pi, \Gamma \Rightarrow \Delta$ is provable in LJ, and similarly for LJ'=.

Then we easily have:

**Corollary 4** A sequent $\Rightarrow A$ is provable in LJ= if and only if it is provable in LJ'=.

**Def 5** Now let $\mathbf{R}$ be a reflexive and transitive relation on the non-empty set $\mathbf{K}$ and let $k \in \mathbf{K}$. A subset $C \subseteq \mathbf{K}$ is a *world-line from* k if C is a maximal subset of $\mathbf{K}$ which is linearly ordered by $\mathbf{R}$ with first element k, and we will write $C \uparrow k$ to indicate this. Also, we write

$$\mathbf{R}_k =_{dfn} \mathbf{R}\text{“}\{k\} =_{dfn} \{k' \in \mathbf{K} : k'\mathbf{R}k\}$$

and

$$\breve{\mathbf{R}}_k =_{dfn} \breve{\mathbf{R}}\text{“}\{k\} =_{dfn} \{k' \in \mathbf{K} : k\mathbf{R}k'\}.$$

**Def 6** A *semi-classical structure* $\mathcal{A}$ for the language L consists of the following entities:

- a non-empty set $|\mathcal{A}|$, the *universe* of $\mathcal{A}$;
- a binary function $\equiv$: $|\mathcal{A}|^2 \rightarrow \{U, V\}$;
- for each n-ary predicate symbol $\mathbf{p}$, an n-ary total function $\mathbf{p}_{\mathcal{A}} : |\mathcal{A}| \rightarrow U, V$;
- for each n-ary function symbol $\mathbf{f}$, where $n > 0$, an n-ary partial function $\mathbf{f}_{\mathcal{A}} : |\mathcal{A}|^n \rightarrow |\mathcal{A}|$;
- for each individual constant $\mathbf{c}$ (i.e., 0-ary function symbol), an individual $\mathbf{c}_{\mathcal{A}} \in |\mathcal{A}|$.

Moreover, we require that for any $a, b, c, \in |\mathcal{A}|$,

- $\equiv (a, a) = v$,
- if $\equiv (a, b) = V$, then $equiv(b, a) = V$, and
- if $\equiv (a, b) = V$ and $\equiv (b, c) = V$, then $\equiv (a, c) = V$.

We will often abbreviate $\equiv (a, b) = V$ by $a \equiv b$; thus $a \equiv b$ is an equivalence relation on $|\mathcal{A}|$. We will generally write

$$\mathcal{A} =< |\mathcal{A}|, \equiv, \mathbf{p}_{\mathcal{A}}, \ldots, \mathbf{f}_{\mathcal{A}}, \ldots, \mathbf{c}_{\mathcal{A}}, \ldots >.$$

The values V and U can be thought of as signifying 'verified' and 'unverified', respectively. The language $\mathrm{L}(\mathcal{A})$ is obtained from L by adding a new individual constant $i_a$ to L for each a$\in |\mathcal{A}|$ ; $i_a$ is called the *canonical name* of a. Then $\mathcal{A}$ has a natural expansion to a semi-classical structure for L(a), namely, set $(i_a)_{\mathcal{A}} = a$ for each $a \in |\mathcal{A}|$.

**Def 7** Given the language L, an *I-structure* A $=< \mathcal{A}_k, \mathbf{K}, \mathbf{R} >$ for L consists of a reflexive and transitive relation $\mathbf{R}$ on a non-empty set $\mathbf{K}$ together with semi-classical structures for L,

$\mathcal{A}_k =< |\mathcal{A}_k|, \equiv_k, \mathbf{P}_k, \ldots, \mathrm{f}_k, \ldots, \mathrm{c}_k >$, such that for all $k \in \mathbf{K}$ (where we write $\mathbf{P}_k$ for $\mathbf{P}_{\mathcal{A}_k}$, etc.) each of the following hold:

1. if $k\mathbf{R}k'$, then $|\mathcal{A}_k| \subseteq |\mathcal{A}_{k'}|$;

2. if $k\mathbf{R}k'$ and $a, b \in |\mathcal{A}_k|$, then $a \equiv_k b$ implies $a \equiv_{k'} b$;

3. if $kRk'$ and $a_1, \ldots, a_n \in |\mathcal{A}_k|$, then $p_k(a_1, ..., a_n) = V$ implies $p_{k'}, (a_1, ..., a_n) = V$.

4. if $k\mathbf{R}k'$, then $dom(\mathbf{f}_k) \subseteq dom(\mathbf{f}_{k'})$ and $f_{k'} \cap dom(\mathbf{f}_k)^2 = \mathbf{f}_k$:

5. if $k\mathbf{R}k'$, then $\mathbf{c}_k = \mathbf{c}_{k'}$;

6. if $a_1 \equiv_k b_1, \ldots, a_n \equiv_k b_n$, and if $< a_1, ..., a_n >$ and $< b_1, ..., b_n >$ are both in $dom(\mathbf{f}_k)$, then

   $\bigwedge k' \in \check{\mathbf{R}}_k \bigvee k'' \in \check{\mathbf{R}}_{k'}[\mathbf{f}_{k''}(a_1, \ldots, a_n) \equiv_{k''} \mathbf{f}_{k''}(b_1, ..., b_n)]$;

7. if $a_1 \equiv_k b_1, \ldots, a_n \equiv_k b_n$, and if $\mathbf{p}(a_1, ..., a_n) = v$, then

   $\bigwedge k' \in \check{\mathbf{R}}_k \bigvee k'' \in \check{\mathbf{R}}_{k'} \mathbf{p}_{k''}(b_1, \ldots, b_n) = V$.

8. $\bigwedge k \bigwedge a_1, \ldots, a_n \in |\mathcal{A}_k| \bigwedge k' \in \check{\mathbf{R}}_k \bigvee k'' \in \check{\mathbf{R}}_{k'}[< a_1, \ldots, a_n > \in dom(\mathbf{f}_{k''})]$.

We will always assume that

$$\mathbf{K} \cap \cup_{k \in \mathbf{K}} |\mathcal{A}_k| = \emptyset.$$

For $S \subseteq \mathbf{K}$, set

$$U(S) =_{dfn} \cup_{k \in S} |\mathcal{A}_k|,$$

where $U(\mathrm{A}) = U(\mathbf{K})$. Let Var be the set of free variables of L ; we will use $\mathbf{x}$,$\mathbf{y}$,$\mathbf{z}$,... to range over Var.

**Def 8** An *assignment* in A is a map $\nu : Var \longrightarrow U(\mathrm{A})$.

**Def 9** If $\nu$ is an assignment in A, $\mathbf{x} \in Var$, and $a \in U(\mathrm{A})$, we define $\nu \begin{pmatrix} \mathbf{x} \\ a \end{pmatrix}$ by

$$\nu \begin{pmatrix} \mathbf{x} \\ a \end{pmatrix} (\mathbf{y}) = \begin{cases} a & \text{if } \mathbf{x} \text{ is } \mathbf{y} \\ \nu(\mathbf{x}) & \text{otherwise} \end{cases}$$

Also, set

$$\nu^{\#}(\mathbf{A}) =_{dfn} \{\mathbf{x}) : \mathbf{x} \text{ is free in } \mathbf{A}\}.$$

**Def 10** Let a be a term. The *denotation of* a *in* A *at* $k \in \mathbf{K}$ *relative to an assignment* $\nu$ is given recursively:

$$\mathrm{x}^{\mathrm{A},k}[\nu] \simeq \begin{cases} \nu(\mathbf{x}) & \text{if } \nu(\mathbf{x}) \in |\mathcal{A}_k| \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathrm{f}_k(a_1,\ldots,a_n)^{\mathrm{A},k}[\nu] \simeq \begin{cases} \mathrm{f}_k(\mathrm{a}_1^{\mathrm{A},k}[\nu],\ldots,\mathrm{a}_n^{\mathrm{A},k}[\nu]) & \text{if } \mathrm{a}_\mathrm{i}^{\mathrm{A},k}[\nu] \in |\mathcal{A}_k| \text{ for i} = 1,\ldots,\mathrm{n} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that if $\mathrm{a}^{\mathrm{A},k}[\nu]$ is defined, it lies in $|\mathcal{A}_k|$.

**Def 11** Given an assignment $\nu$ in A, $k \in \mathbf{K}$, and a formula $\mathbf{A}$ , we define a satisfaction operator $\mathrm{A}^{k,\nu}$ by recursion as follows (recall that V = 'verified' and U = 'unverified'):

1. $\mathrm{A}^{k,\nu}(\mathbf{a} = \mathbf{b}) = \begin{cases} \mathrm{a}^{\mathrm{A},k}[\nu] \equiv_k \mathrm{b}^{\mathrm{A},k}[\nu] & \text{if } \mathrm{a}^{\mathrm{A},k}[\nu], \mathrm{b}^{\mathrm{A},k}[\nu] \text{ are both defined} \\ U & \text{otherwise} \end{cases}$

2. $\mathrm{A}^{k,\nu}(\mathrm{pa}_1\ldots\mathrm{a}_n) = \begin{cases} \mathrm{p}_k(\mathrm{a}_1{}^{\mathrm{A},k}[\nu],\ldots,\mathrm{a}_n{}^{\mathrm{A},k}[\nu]) & \text{if } \mathrm{a}_\mathrm{i}{}^{\mathrm{A},k}[\nu] \text{ defined for i} = 1,\ldots,\mathrm{n} \\ U & \text{otherwise} \end{cases}$

3. $\mathrm{A}^{k,\nu}(\mathbf{A} \wedge \mathbf{B}) = \begin{cases} V & \text{if } \mathrm{A}^{\mathrm{k},\nu}(\mathbf{A}) = \mathrm{A}^{\mathrm{k},\nu}(\mathbf{B}) = \mathrm{V} \\ U & \text{otherwise} \end{cases}$

4. $\mathrm{A}^{k,\nu}(\mathbf{A} \vee \mathbf{B}) = \begin{cases} V & \text{if } \wedge \mathrm{C} \uparrow \mathrm{k} \vee \mathrm{k}' \in \mathrm{C}[\mathrm{A}^{\mathrm{k}',\nu}(\mathbf{A}) = \mathrm{V} \text{ or } \mathrm{A}^{\mathrm{k}',\nu}(\mathbf{B}) = \mathrm{V}] \\ U & \text{otherwise} \end{cases}$

5. $\mathrm{A}^{k,\nu}(\neg\mathbf{A}) = \begin{cases} V & \text{if } \mathrm{A}^{\ell,\nu}(\mathbf{A}) = \mathrm{U} \text{ for all } \ell \in \breve{\mathbf{R}}_{\mathrm{k}} \\ U & \text{otherwise} \end{cases}$

6. $\mathrm{A}^{k,\nu}(\mathbf{A} \rightarrow \mathbf{B}) = \begin{cases} V & \text{if } \mathrm{cond}_1(\mathbf{A}, \mathbf{B}) \\ U & \text{otherwise} \end{cases}$

7. $\mathrm{A}^{k,\nu}(\forall \mathrm{x}\mathbf{A}) = \begin{cases} V & \text{if } \mathrm{cond}_2(\mathrm{x}, \mathbf{A}) \\ U & \text{otherwise} \end{cases}$

8. $\mathrm{A}^{k,\nu}(\exists \mathrm{x}\mathbf{A}) = \begin{cases} V & \text{if } \mathrm{cond}_3(\mathrm{x}, \mathbf{A}) \\ U & \text{otherwise} \end{cases}$

where we use the following abbreviations:

$cond_1(\mathbf{A}, \mathbf{B})$ iff:

$\wedge C \uparrow k \vee k' \in C \wedge \ell \in C[\nu^{\#}\mathbf{A} \subseteq U(C)\ \&\ \mathrm{A}^{\ell,\nu}(\mathbf{A}) = V \implies \vee m \in C \cup \breve{\mathbf{R}}_{\ell}[\mathrm{A}^{m,\nu}(\mathbf{B}) = V]]$;

$cond_2(\mathrm{x}, \mathbf{A})$ iff:

$$\wedge C \uparrow k \vee k' \in C \wedge \ell \in C \wedge a \in |\mathcal{A}_{\ell}| \wedge m \in C \cap \breve{\mathbf{R}}_{\ell}[\nu^{\#}\mathbf{A} \subseteq U(C) \implies \mathrm{A}^{m,\nu}\begin{pmatrix} \mathrm{x} \\ a \end{pmatrix}(\mathbf{A})]$$

$cond_3(\mathrm{x}, \mathbf{A})$ iff:

$$\wedge C \uparrow k \vee k' \in C[\mathbf{A} \subseteq C \implies \wedge \ell \in C \wedge a \in |\mathcal{A}_{\ell}|[\mathrm{A}^{\ell,\mu}\begin{pmatrix} \mathrm{x} \\ a \end{pmatrix}(\mathbf{A})]].$$

Note that the mapping $\mathrm{A}^{k,\nu}(\mathbf{A})$ is always defined. We say that $\mathbf{A}$ is *valid in* A if

for each assignment $\nu$ in A and each world-line C in A (i.e., maximal subset of **K** linearly ordered by **R**) such that $\nu^{\#}\mathbf{A} \subseteq U(C)$, there is a $k \in C$ such that $\mathrm{A}^{k,\nu}(\mathbf{A}) = V$. A sequent $\mathbf{A}_1, \ldots, \mathbf{A}_n \Rightarrow \mathbf{B}_1, \ldots, \mathbf{B}_m$ is *valid in* A if and only if the formula

$$\mathbf{A}_1 \wedge \ldots \wedge \mathbf{A}_n \rightarrow \mathbf{B}_1 \vee \ldots \vee \mathbf{B}_m$$

is valid in A. A formula or sequent is *valid* if and only if it is valid in all I-structures. The following lemmas are easy to verify by induction.

**Lemma 2** Let **A** be a formula, let **b** and **c** be terms, let $k \in \mathbf{K}$, and let $\nu$ be an assignment in A. Then:

1. $(\mathbf{b}_{\mathbf{x}}[\mathbf{a}])^{\mathrm{A},k}[\nu] \simeq \mathbf{b}^{\mathrm{A},k}[\mu \begin{pmatrix} \mathbf{x} \\ \mathbf{a}^{\mathrm{A},k}[\nu] \end{pmatrix}]$;

2. $\mathrm{A}^{k,\nu}(\mathbf{A}_{\mathbf{x}}[\mathbf{a}]) = \mathrm{A}^{k,\nu}\begin{pmatrix} \mathbf{x} \\ \mathbf{a}^{\mathrm{A},k}[\nu] \end{pmatrix}(\mathbf{A})$.

**Lemma 3** Let **A** be a formula, let **a** be a term, let $k, k' \in \mathbf{K}$, and let $\nu$ be an assignment in A. Then:

1. if k**R**k' and $\mathbf{a}^{\mathrm{A},k}[\nu]$ is defined, then $\mathbf{a}^{\mathrm{A},k'}[\nu]$ is defined and $\mathbf{a}^{\mathrm{A},k}[\nu] = \mathbf{a}^{\mathrm{A},k'}[\nu]$;

2. if k**R**k' and $\mathrm{A}^{k,\nu}(\mathbf{A}) = V$, then $\mathrm{A}^{k',\nu}(\mathbf{A}) = V$.

## Main Theorems

**Theorem 12 (Validity Theorem)** If $\Gamma \Rightarrow \Delta$ is provable in $LJ'_{=}$, then it is valid.

**Theorem 13 (Completeness)** A sequent $\Gamma \Rightarrow \Sigma$ is provable in $LJ'_{=}$ without cut iff it is valid in all I-structures in which every $|\mathcal{A}_k|$ is finite.

References [1] Bowen, K.A., *A note on cut elimination and completeness in first order theories*, **Zeit. F. math. Logik und Grund. d. Math., 18** (1972), 173-176.

[2] Gentzen, G. *Investigations into logical deduction*, in **The Collected Papers of Gerhard Gentzen** Amsterdam 1969, 68-131.

[3] Prawitz, D. *Some results for intuitionistic logic with second order quantification rules*, in **Intuitionism and Proof Theory**, Amsterdam, 1970, 259-269.

[4] Shoenfield, J. **Mathematical Logic**, Reading, Mass., 1967.

### 11.6.2 Theoretical Semantics

The logic language on which metaProlog is based amalgamates an object-level language with its metalanguage. Consequently, attempts to adapt standard Tarskian semantics are extremely ugly, and semantic interpretations based on Kripkian semantics are only slightly better. A much more promising approach has arisen by taking two moves. First (related to the approach of A. Church's Theory of Types) is to regard *all* the expressions of the language as terms, with the formulas being merely a distinguished subclass of the terms. Second, one abandons the normal two-value truth value set and the normal "set of individuals" for the construction of denotations, and replaces them jointly with the set of all "reasonable" syntactic entities from the language itself, including partial proofs and search spaces. One then constructs a semantic interpretation using the so-called "substitutional interpretation", but attaches collections of partial proofs and partial search spaces to pairs of theories and formulas (the latter regarded as goals to be solved in the theory). It appears that many of the basic theorems of standard logic programming theory can be pushed through by brute force. However, it is much more appealing to attempt to adapt the ideas of Blair, Brown, and Subramanian which have shown that the basic theorems can be proved abstractly, given a suitable lattice structure on the space of truth values. The next step is the search for such a suitable lattice structure on the space of syntactic entities.

### 11.6.3 Reason Maintenance Experiment

We have been exploring several experimental knowledge-base management systems implemented using the metaProlog compiler(s). We exhibited a typical example at the RADC/NAIC Technology Fair during April, 1987. The top level of the system is sketched below. The primary predicates are

```
kbm(kb, int, mnt, kb_time)

react_to(request, kb, int, mnt, kb_time)
```

which are mutually tail-recursive. Three of the arguments are theories:

- kb – the domain knowledge base
- int – the theory defining integrity and consistency of kb
- mnt – they theory containing rules for revision and maintenance, together with data to effect revisions, etc.

The argument 'kb_time' simply is an abstract clock representation (here, system cycles - it could be real time). And the argument 'request' is simply the request for action obtained from the user: a query, a requested update, etc.

```
/*-----------------------------------------------------------------
       knowledge base manager main loop
        -- kbm and react_to are mutually tail-recursive
-----------------------------------------------------------------*/

all [kb, int, mnt, request, kb_time] :
   kbm(kb, int, mnt, kb_time)
   <-
     get_req(kb, request, kb_time) &
     react_to(request, kb, int, mnt, kb_time).
```

```
/*-------------------------------------------------------------------
          kbm primary action predicate: react_to
-------------------------------------------------------------------*/
/*===================================================
          Handling queries
     -- this simply returns one solution
         (prints the instantiated query)
     multiple solutions are handled by the "all [..." request below

===================================================*/

all [kb, int, mnt, question, kb_time]  :
   react_to(query(question), kb, int, mnt, kb_time)
   <-
     demo(kb, question) & ! &
     write('<<kbm: ') & write(question) & nl &
     kbm(kb, int, mnt, kb_time).

all [kb, int, mnt, question, kb_time]  :
   react_to(query(question), kb, int, mnt, kb_time)
   <-     ! &
     write('<<kbm--No solution: ') & write(question) & nl &
     kbm(kb, int, mnt, kb_time).

/*===================================================================
     Queries requesting all solutions
     Input form is:
        all [x,y,z,...] : Formula
===================================================================*/

all [kb, int, mnt, question, kb_time, vars, form, vars,
     realVars, instantiatedForm, sols, numVars]  :
   react_to(all(vars, form), kb, int, mnt, kb_time)
   <-     ! &
     write('Trying all sols...') & nl &
     length(vars, numVars) &              %create Prolog vars
```

```
    make_var_list(numVars, realVars) &
    subst_prolog(form, vars, realVars, instantiatedForm) &
         %instantiate Formula
    (demo(kb, setof(realVars, instantiatedForm, sols)) & ! &
     write('Solutions found:') & nl &
     show_list(sols);
       %% No solutions alternative
        write('No solutions found...') & nl)
    & kbm(kb, int, mnt, kb_time).

/*============================================================
    React to requests to add an assertion to the kb
============================================================*/

all [kb, int, mnt, assertion, vars, var_list, form, inst_form, new_mnt,
         new_kb, kb_time, new_kb_time]  :
  react_to(add(assertion), kb, int, mnt, kb_time)
  <-     ! &
    (assertion = ':'(all(vars), form) & ! &
     instantiate(assertion, inst_form, var_list) &
     check_conseqs(inst_form, var_list, kb, int, mnt,
              new_kb, new_mnt, assertion, kb_time);
         integ_ck(assertion, kb, int, mnt, new_kb, new_mnt, ko_time)) &
    new_kb_time is kb_time + 1 &
    kbm(new_kb, int, new_mnt, new_kb_time).
```

The subsidiary predicates of interest above are those dealing with the checking of consequences, integrity, and consistency, defined as follows.

```
all [form, var_list, kb, int, mnt, new_kb, new_mnt, assertion,
                        inst_form, kb_time] :
  check_conseqs(inst_form, var_list, kb, int, mnt,
                    new_kb, new_mnt, assertion, kb_time)
  <-
    (work_thru_conseqs(inst_form, var_list, kb, int, mnt,
                          new_kb, new_mnt, assertion, kb_time) & !;
       write('Denying addition of assertion to the kb...') & nl &
```

```
        new_kb = kb &
        addto(mnt, failed_add(assertion,kb,int,mnt,kb_time), new_mnt)).


all [inst_form, kb, int, mnt, new_kb, new_mnt, assertion, kb_time,
     head, body, exist_quant_body, var_list, head_list, head_vars,
     body_vars] :
   work_thru_conseqs(inst_form, var_list, kb, int, mnt,
              new_kb, new_mnt, assertion, kb_time)
   <-
     (inst_form = (head :- body) & ! &
          vars_occurring_in(head, head_vars) &
          difference(var_list, head_vars, body_vars) &
          exist_quant(body_vars, body, exist_quant_body) &
       (demo(kb, setof(head, exist_quant_body, head_list)) &
            show_list(head_list,3);
        write('No immediate head consequences of this assertion...') &nl);
          write('The expression ') & write(inst_form) &
          write(' is not an implication...ignoring for now...') &nl) &
     addto(mnt, added(assertion, kb, int, mnt, kb_time), new_mnt) &
     addto(kb, inst_form, new_kb).


all [kb, int, mnt, assertion, new_kb, new_mnt, kb_time] :
   integ_ck(assertion, kb, int, mnt, new_kb, new_mnt, kb_time)
   <-
     (demo(kb, assertion) & ! & write('Duplication--nothing added') & nl ;
        demo(int+kb, acceptable(assertion)) &
        contradict_check(assertion, kb, int) &
        write('Integrity check passed...') & nl &
        addto(mnt, added(assertion, kb, int, mnt, kb_time), new_mnt) &
        addto(kb, assertion, new_kb)).

all [kb, int, mnt, assertion, new_kb, new_mnt, kb_time, body,
              ans, ans1, inst_body,d,e,f,g] :
   integ_ck(assertion, kb, int, mnt, new_kb, new_mnt, kb_time)
   <-
```

```
    not(demo(int,clause(acceptable(assertion), body))) &
    (demo(int+kb, update_via(assertion, body)) &
     write('Assertion to add is defined by the following view:')
     & nl & nl &
     write('   ') & write(assertion) &
     write(' if ') & write(body) & put(~.) & nl & nl &
     write('Do you want to attempt the addition via this view?') &
     read(ans1) &
     (affirmative(ans1) & ! &
        (not(body = (d,e)) &
         demo(int+kb, acceptable(body)) & !; true) & ! &
         (not(body = (f,g)) &
          contradict_check(assertion, kb, int) & !; true) &
            write('Integrity check passed...') & nl &
         check_instances(body, inst_body),
            addto(mnt,added(inst_body, kb,int,mnt,kb_time), new_mnt) &
            addto(kb, inst_body, new_kb); fail);
                          %otherwise for demo(int+kb, update....)
     contradict_check(assertion, kb, int) &
     write('No (other) integrity clauses apply to ') &
     write(assertion) & nl &
     write('Do you want to accept it as a pure premise?') & read(ans) &
     integ_act_on(ans,assertion,kb,int,mnt,new_kb,new_mnt,kb_time)).

all [kb, int, mnt, assertion, new_kb, new_mnt, kb_time, ans] :
   integ_act_on(ans, assertion, kb, int, mnt, new_kb, new_mnt, kb_time)
   <-
     affirmative(ans) &
     write('Adding premise: ') & write(assertion) & nl &
     addto(mnt, added(assertion, kb, int, mnt, kb_time), new_mnt) &
     addto(kb, assertion, new_kb).

all [kb, int, mnt, assertion, new_kb, new_mnt, kb_time, ans] :
   integ_act_on(ans, assertion, kb, int, mnt, kb, new_mnt, kb_time)
   <-
     not(affirmative(ans)) &
     write('Denying addition of premise: ') & write(assertion) & nl &
```

```
    addto(mnt, failed_add(assertion, kb, int, mnt, kb_time), new_mnt).

all [kb, int, mnt, assertion, new_mnt, kb_time] :
   integ_ck(assertion, kb, int, mnt, kb, new_mnt, kb_time)
   <-
     write('Integrity check failed for ') & write(assertion) &
     write(' at time ') & write(kb_time) & nl &
     addto(mnt, failed_add(assertion, kb, int, mnt, kb_time), new_mnt).

all [assertion, kb, int, contrary] :
   contradict_check(assertion, kb, int)
   <-
     demo(int, contradictory(assertion, contrary)) &
     & demo(kb, contrary)
     & ! & fail.

all [assertion, kb, int] :
   contradict_check(assertion, kb, int).
```

The general knowledge-base management machinery built up above was applied to a small application concerning the NAIC consortium. First we need a starting knowledge base. This could have begun empty. The overall system provides facilities for saving the current state of the kbm system.

```
theory(naic_kb).      % the knowledge

located(su, city(syracuse)).
located(city(syracuse), state(ny)).
located(ub, city(buffalo)).
located(city(buffalo), state(ny)).
located(um, city(amherst)).
located(city(amherst), state(mass)).

pi(person(lesser, vic), project(1)).
pi(person(croft,bruce), project(1)).
title(project(1),[a,knowledge,acquisition,assistance,and,explanation,system]).
```

```
pi(person(bowen,ken), project(2)).
title(project(2),[knowledge,base,maintenance]).

pi(person(shapiro,stu), project(3)).
title(project(3), [a,versatile,expert,system,for,equipment,maintenance]).

located(radc, afb(griffiss)).
located(afb(griffiss), state(ny)).

all [x,y] : in(x,y) <- located(x,y).
all [x,y,z] : in(x,y) <- located(x,z), in(z,y).

all [x,y,z] : same_state(x,y) <- in(x,state(z)) & in(y, state(z)).

all [person, title, project] :
   directs(person, title)
   <-
     pi(person, project) & title(project, title).

endtheory.
```

Next we need definitions of predicates which provide for integrity and consistency maintenance. There are several points worth noting. First, the definition of 'acceptable' has only one argument, namely the proposed new addition to the knowledge. However, examination of the code for 'integ_ck' shows that the goal

```
acceptable(Update)
```

is run in the context of the current state of the knowledge base (combined with the theory 'naic_int' defined below). Thus, this effectively defines the notion of 'acceptable with respect to the current knowledge base'. Secondly, the definitions of 'acceptable' and 'inconsist' are domain-specific: They apply to the anticipated assertions which the system may consider. Finally, note that the knowledge base designer can define procedures (possibly domain-specific) which update derived views under acceptable circumstances, as seen in 'rec_update_via'.

```
theory(naic_int).   meta-level integrity & consistency


all [place1, place2] :
   acceptable(located(place1, place2))
   <-
     subsidiary(place1, place2).

all [place1, place2] :
   subsid(place1, place2) <- atom(place1).

all [place1, place2] :
   subsid(city(place1), state(place2)).

all [place1, place2] :
   subsid(state(place1), country(place2)).

all [place1, place2] :
   subsid(country(place1), continent(place2)).

all [place1, place2] :
   subsidiary(place1, place2)
   <-
     subsid(place1, place2).

all [place1, place2, place3] :
   subsidiary(place1, place2)
   <-
     subsid(place1, place3) & subsidiary(place3, place2).

all [x] :
   is_place(city(x)) <- atom(x).

all [x] :
   is_place(state(x)) <- atom(x).
```

```
all [x] :
   is_place(country(x)) <- atom(x).

all [x] :
   is_place(continent(x)) <- atom(x).

all [name1, name2, number, boss, what] :
   acceptable(pi(boss, what))
   <-
     boss = person(name2, name1)
     & what = project(number) & integer(number).

all [number, words] :
   acceptable(title(project(number), words))
   <-
     integer(number) & list_of_atoms(words).

list_of_atoms([]).

all [head, tail] :
   list_of_atoms([head | tail])
   <-
     atom(head) & list_of_atoms(tail).

%%========= Rules for inconsistency ====== %%%%%

all [x, y] :
   contradictory(x, y)
   <-
     inconsist(x, y).

all [x, y] :
   contradictory(x, y)
   <-
     inconsist(y, x).

all [x] :
```

```
   inconsist(male(x), female(x)).

all [x] :
   acceptable(male(x)).

all [x] :
   acceptable(female(x)).

%%===== Rules for updating views ==== %%%%%

all [assertion, definition, body] :
   update_via(assertion, definition)
   <-
     clause(assertion, body) &
     rec_update_via(body, definition).

all [a1,a2, b1,b2] :
   rec_update_via((a1,a2), (b1, b2))
   <-   ! &
     rec_update_via(a1, b1) & rec_update_via(a2, b2).

all [assertion, definition] :
   rec_update_via(assertion, definition)
   <-
     var(assertion) & ! & fail.

all [assertion, definition] :
   rec_update_via(assertion, definition)
   <-
     atom(assertion) & ! &
     (clause(assertion, body) & ! &
      rec_update_via(body, definition) ;
          definition = assertion).

all [assertion, predicate, args] :
   rec_update_via(assertion, assertion)
   <-
```

```
    assertion =.. [predicate | args].

endtheory.

theory(naic_mnt).     % kbm maintenance

all [x,y] :
   subset(x, y)
   <-
     subset0(x,y).

all [x,y,z] :
   subset(x, y)
   <-
     subset0(x,z) & subset(z, y).

all [x, y, z] :
   belongs(x,y)
   <-
     belongs_to(x, y, z).

all [x, y, z, u] :
   belongs(x, y)
   <-
     true &
     subset0(u, y) &
     demo(y, clause(x, z)).

endtheory.
```

While only a small toy example, the code above demonstrates the ease with which knowledge base implementers can directly define notions of consistency and maintance specific to the content of the particular application. (No revision maintenance is conducted in this example. See the 'ltm' example below.)

### 11.6.4 Reason Maintenance and Theory Manipulation

By bringing the problem-solver and knowledge-base maintenance program closer together than previously, a very useful and potentially efficient methodology has evolved. As in the preceeding section, one writes a version of the metaProlog interpreter as normally expressed in Prolog with an explicit argument indicating the theory underwhich the deduction is being performed. However, in this interpreter, one includes an *implementation* of the usual Prolog assert. At the metalevel, this is a logical axiomatization of such a system. The implementation of assert requires that the interpreter check the consistency of the assertion being added against an integrity theory which is also carried around by the interpreter. If the consistency check fails, the interpreter consults its revision theory to guide revision of the knowledge base to a consistent state. As is usual with such interpreters, a source-to-source transformer is created which partially evaluates the domain problem-solving rules and knowledge base relative to this interpreter (via expansion of arguments). Not only do the transformed rules run much more efficiently, but consideration of the manner in which they are compiled provides insight into how the process might be pushed deeper into the metaProlog compiler's abstract machine.

Some exploratory work on an implementation (in metaProlog) of a "logic-based" reason maintenance system in the style of McAllester was begun. A top-level sketch of this experiment follows. (It is written in conventional Prolog syntax which our metaProlog compiler accepts. When more fully developed, it will be converted to metaProlog syntax via an automatic conversion program.)

```
/*-------------------------------------------------------------------
 |          ltm.pro
 |          Logic-Based Truth Maintenance
 *-------------------------------------------------------------------*/

solve(Problem, Solution, KB, Final_KB)
   :-
   demo(solver, solved(KB, Solution^Problem) ).

x(N) :-
   name(N, N_String),
   append("examp", N_String, ".pro", File_String),
```

```
   name(File, File_String),
   append("x", N_String, Th_Name_String),
   name(Theory_Name, Th_Name_String),
   consult(File, Theory_Name),
   demo(Theory_Name, initialize_kb(KB) ),
   demo(Theory_Name, goal_problem(Problem/Solution) ),
   solve(Problem, Solution, KB, Final_KB),
   nl, write(problem=Problem),nl,
   write(' solved by solution:'),nl,
   write(Solution),nl.


theory solver. % $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

use(kb_primitives).

solved(KB, Output^Problem, T) :-
   current_focus(KB, T0),
   solves(Problem, KB, T0, T).

solves(Problem, KB, KB, T, T)
   :-
   status(Problem, KB, T, true), !.

solve( (Prob1 & Prob2), KB0, KB1, T0, T1)
   :- !,
   solves(Prob1, KB0, KB_Inter, T0, T_Inter),
   solves(Prob2, KB_Inter, KB1, T_Inter, T1).

solves(Problem, KB0, KB1, T0, T1)
   :-
   rule_of(KB0, T0, Problem, Body),
   solves(Body, KB0, KB1, T0, T1).

solves(Problem, KB0, KB1, T0, T1)
   :-
   status(Problem, KB0, T0, unknown),
```

```
   possible_assumption(KB0, T0, Problem),
   demo(reason_maint, assumable(Problem, KB0, KB1, T0, T1) ).

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  Deep failure causing backtracking to this point will be handled
  tail-recursively inside 'reason_maint' in the definition of
  assumable, which will look for an acceptable way to back up
  the theory T0 (typically removing some assumptions under some
  maintenance regime) to yield a theory T3 and knowledge base
  state KB3, and then calling
solves(Problem, KB3, KB1, T3, T1).
 Consequently, we see that if we originally submit the goal
:-solves(Problem, KB0, KB1, T0, T1)
 and it succeeds, T1 is not necessarily a monotonic extension of T0,
 but is an extension of some acceptable revision of T0.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

endtheory. % solver ##########################################

theory reason_maint. % $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

use(kb_primitives).

assumable(Formula, KB0, KB1, T0, T1)
   :-
   status( not(Formula), KB0, T0, true), !,
   find_alternate(Formula, KB0, T0, KB3, T3),
   demo(solver, solves(Formula, KB3, KB1, T3, T1) ).

assumable(Formula, KB0, KB1, T0, T1)
   :-
   integrity_theory_of(KB0, Integ_Th),
   demo(Integ_Th, acceptable(Form, T0) ),
   not( inconsistent(Formula, T0, KB0) ),
   addto(T0, Formula, T1),
   update_maint_records(T0, addto(T0, Formula, T1), T1, KB0, KB1).
```

```
assumable(Formula, KB0, KB1, T0, T1)
   :-
   find_alternate(Formula, KB0, T0, KB3, T3),
   solves(Formula, KB3, KB1, T3, T1).

inconsistent(Formula, Theory, KB)
   :-
   status(Formula, Other_Theory, KB, false),
   records(KB, extends(Other_Theory, Theory) ), !.

inconsistent(Formula, Theory, KB)
   :-
   demo(Theory, not(Formula) ).

endtheory. % reason_maint ######################################

theory kb_primitives. % $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

records(KB, extends(T1, T2) )
   :-
   extension_records_of(KB, Ext_Recs),
   demo(Ext_Recs, extends(T1, T2) ).

update_maint_records(T0, addto(T0, Formula, T1), T1, KB0, KB1)
   :-
   extension_records_of(KB0, Ext_Recs0),
   addto(Ext_Recs0, extends(T0, T1), Ext_Recs1),
   update_kb(extension_records, Ext_Recs1, KB0, KB1).

status(Formula, Theory, KB, Status_Value)
   :-
   kb_access(status_records, KB, Status_Theory),
   demo(Status_Theory, status(Theory, Status_Value) ).

extension_records_of(KB, Ext_Recs)
   :-
   kb_access(extension_records, KB, Ext_Recs).
```

```
current_focus(KB, T)
   :-
   kb_access(current_focus, KB, T).

kb_access(What, KB, Ext_Recs)
   :-
   kb_access_table(What, ArgNum),
   arg(ArgNum, KB, Ext_Recs).

kb_vector_size(3).

kb_access_table(status_records, 1).
kb_access_table(extension_records, 2).
kb_access_table(current_focus, 3).

make_kb(Arg_List, KB)
   :-
   kb_vector_size(KB_Size),
   functor(KB, kb, KB_Size),
   install_kb_args(Arg_List, KB).

install_kb_args([], KB).

install_kb_args([ Entry_Name = Entry_Value | Rest_Arg_List], KB)
   :-
   kb_access_table(Entry_Name, Entry_Num),
   arg(Entry_Num, KB, Entry_Value),
   install_kb_args(Rest_Arg_List, KB).

endtheory. % kb_primitives ########################################
```

In the course of working on the ltm example, it is becoming clear that our approach to metaProlog may provide an even greater potential with regard to reason maintenance than we originally thought. When one reflects on the details of our metaProlog

compiler, one sees that there are two distinct components to the treatment of theories: (1) provision for the raw physical notion of individual independent theories; in our system, theories are identified with clause-indexing patches referring to subsets of a global "blackboard" of clauses; (2) maintenance of relationships among theories; in the present metaProlog, we support maintenance of historical relationships.

The implementation of the former is independent of the latter (though 2 does rely on 1, but this causes no problem for the following). It seems apparent that (2) could be replaced or supplemented by maintenance of other sorts of relationships between theories, in particular, the sorts of relationships inherent in reason maintenance.

A good deal of more exploration and experimentation will be necessary before the situation becomes sufficiently clear to determine whether the processes of reason maintenance can be sufficiently analyzed into primitive process to warrant elaboration of additional instructions and facilities in the underlying Abstract Prolog Machine.

Assumption-Based Reason Maintenance was also a concern. Here the primary concern is with the excessive storage demands of de Kleer's methods. The goal is to discover methods of achieving much more virtual implementations of his ideas. The target is to be able to explicitly maintain the theories lying along the two fringes: The boundary between the unexamined theories and the known inconsistent theories, and the boundary between the unexamined theories and the known consistent theories. All other theories which have been examined (both consistent and inconsistent) will be maintained in a compact a virtual representation as possible by describing them in terms of theories lying on the fringe. In essence, these descriptions say what must be added to or deleted from a fringe theory in order to obtain a given virtual theory. The advantage can be gained by grouping the theories around the statements. Conceptually, the maintenance will involve quadruples

v(Formula, Sign, FringeTheory, L),

where L is a list of virtual theory ids such that the given formula must be added to (Sign = +) or deleted from (Sign = -) FringeTheory for each of the theories whose id is on the list.

## 11.7 Ilyas Cicekli

### *The Design and Implementation of The metaProlog System*

Most of the meta-level systems implemented in the last decade are meta-level interpreters which introduce extra interpretation layers that slow down the execution. The metaProlog system described in this report is a compiler-based meta-level system for the metaProlog programming language. Since metaProlog is an extension of Prolog, we extended the Warren Abstract Machine (WAM) to the Abstract metaProlog Engine (AMPE). metaProlog programs are directly compiled into the instructions of the AMPE.

In the rest of this report, the metaProlog system is briefly described. Theories which are first class objects in metaProlog, and their representations in the metaProlog system are discussed in Section 2. The basic structure of the AMPE is explained in Section 3. In the last section, the garbage collector of the metaProlog system is presented.

### 11.7.1 metaProlog Theories

In Prolog, there is a single database, and all goals are proved with respect to this database. When there is a need to update this database, the builtins assert/retract, which are ad hoc extensions to the basic logic programming paradigm, are used to create the new version of this database by destroying the old database in the favor of the new one. On the other hand, there can be more than one theory in metaProlog, and a goal can be proved with respect to one of these theories. A new theory in metaProlog is created from an old theory without destroying the old theory.

A new theory is created from an old theory that already exists in the system by adding some clauses or dropping them. The new theory inherits all procedures of the old theory except procedures explicitly modified during its creation. Although we create a new theory from an old theory, the old theory is still accessible by the user.

The provability relation between a theory and a goal is explicitly represented in metaProlog by a two argument predicate "demo". The relation "demo(Theory,Goal)" precisely holds when "Goal" is provable in "Theory". Similarly, the relation *demo*( *Theory*, *Goal*, *Proof*) holds when "Proof" is the proof of "Goal" in "Theory". When one of these provability relations is encountered, the underlying theorem prover tries

to prove the given goal with respect to the given theory.

Theories of the metaProlog system are organized in a tree whose root is a distinguished theory, the base theory. The base theory contains all the system builtins, and all other theories in the system are descendants of the base theory. In other words, all theories can access procedures of the base theory.

Every theory in the metaProlog system possesses a default theory except for the base theory. The default theory of a theory T is the theory where we search for a procedure if the search for that procedure in T fails. This search through default theories continues until the procedure is found or the base theory is reached.

To shorten the depth of the theory, theories in the metaProlog system are classified into two groups : "default theories", and "non-default theories". A "non-default theory" is a theory that carries information about all procedures that underwent modifications in the ancestor theories between this theory and its default theory. Access to these procedures is very fast, at the expense of copying some references. The default theory of a theory is the first ancestor theory that is a "default theory". A "default theory" is a theory whose descendants don't carry any information about th procedures occurring in that theory. If only default theories are used, access to a given procedure in a given theory may require a search through all its ancestor theories. In this case, access to a procedure may be slow, but no copying of references is needed. Depending on the problem, the system tries to use one or the other approach, or a combination of both to achieve a balance between speed of access and space overhead.

When a new theory is created from a non-default theory, its default theory will be its father's default theory. But if a new theory is created from a default theory, its default theory will be its father. In the first case, the new theory will be at its father's level. In the second case, the new theory will be at one level above its father's level. Thus we don't increment the depth of the theory tree when a theory is created from a non-default theory.

### 11.7.2 Abstract metaProlog Engine

Our main goal in this project was to create an efficient compiler-based metaProlog system. Since metaProlog is an extension of Prolog, the Warren Abstract Machine (WAM) was the best starting point. For this purpose, the WAM is extended to the Abstract metaProlog Engine (AMPE).

The AMPE performs most of the functions of the WAM, but it also has some extra features to handle theories and compiled procedures as data objects of the system. These extra features basically are:

- Extra registers to handle theories in metaProlog.
- A different memory organization which is more suitable to handle compiled procedures and theories as data objects of the system.
- The functions of the procedural instructions in the AMPE differ from their functions in the WAM.

There are two new registers in the AMPE in addition to the registers the WAM does. The first one is the "theory register" which holds the current theory (context) of the metaProlog system. The value of the "theory register" is changed when the context of the system is switched to another context. This register is also saved in choice points so that the context of the system can be restored the value saved in the last choice point during backtracking. The second one is the "theory counter register" which is simply a counter to produce a unique theory-id for each theory in the system. It is is incremented to indicate the next available theory-id after the creation of each theory.

The code space and the heap in the WAM are integrated as a single data area in the AMPE which is more suitable to handle compiled procedures as data objects. This integrated space in the AMPE is still called "heap". Thus theories and compiled procedures can be created on fly, and they are can be easily discarded when the need for them is gone. The local stack and the trail of the AMPE still perform the same job they perform in the WAM.

### 11.7.3 Proofs

The AMPE can run in two different modes. When a two argument "demo" predicate is encountered, the system runs in the simple mode. In the simple mode, the system only proves a goal with respect to the current theory of the system. When a three argument "demo" predicate is encountered, the mode of the system is switched to the proof mode. In the proof mode, a goal is not only proved with respect to the current theory of the system, its proof is also collected. At the implementation level, the

mode of the system is represented by a mode flag which is also saved in choice points so that the system can switch from one mode to the another during backtracking.

In the simple mode of the system, only the core part of the system described above is used. On the other hand, two extra registers are used in addition to the core part of the system when the system runs in the proof mode. These extra two registers are used to collect the proof of a goal during its execution.

### 11.7.4 Fail Branches

After finishing the core part of the metaProlog system, I started to extend the metaProlog system which can handle extra control information in the demo predicate. Now, the metaProlog system have the following capabilities.

1. Now the system can get fail branches of a goal in addition to its success branches (proofs). When the goal "demo(T,G,branch(P))" is submitted, P is unified with a branch (fail or success) of the proof tree of G in T. On the other hand, when the goal "demo(T,G,proof(P))" is submitted, P is unified with only a success branch of the proof tree of G in T.

2. The system also supports a fourth argument demo whose fourth argument is control information. In some cases, to get a complete proof of a goal can be unnecessary. We may not need all proofs of subgoals. For this purpose, proofs of these subgoals can be skipped by using the following form of the demo predicate.

$$demo(T, G, proof(P), skip\_proofs\_of(Subgoals))$$

   After the execution of the above, proofs of SubGoals don't appear in the proof P of G in T.

### 11.7.5 Garbage Collector

The garbage collector of the metaProlog system collects all the garbage in the system including the garbage in the code. It consists of a recursive marking routine and a compaction routine. The marking routine recursively marks all locations in the

heap which are accessible from external locations such as argument registers, and locations in the local stack. The garbage compaction routine, an extension of Morris's compaction algorithm, adjusts all pointers in the uncompacted heap and does the real compaction.

## 11.8 Keith Hughes[1]

*Interfaces to Databases*

### 11.8.1 Introduction

The combination of logic programming and relational database systems is a desirable goal, because intelligent processing of large numbers of facts becomes possible. Database systems are very good at retrieving large amounts of data while doing little or no inference. On the other hand, logic programming languages such as Prolog provide powerful methods for doing inference, but are inadequate when it comes to processing substantial bodies of facts.

Logic and relational database systems (RDBS) are known to have close theoretical connections [Gallier78]; and many people have advocated an amalgam of the two. Extensions to Prolog to achieve such an amalgamation have been suggested, but there are problems with each. They do, however, point to possible solutions. VMProlog allows SQL queries to be used in the middle of Prolog statements, but this makes a distinction between program and data. The resulting programs are overly complicated. Other methods, which require direct modifications to Prolog itself, include the **compiled method** [Reiter78a] and the **interpretive method** [Minker78].

The system to be described here is a combination of a Prolog system and the RDBS system Ingres [Stonebraker76]. This system provides a framework for experimentation with alternatives for handling the interface between Prolog and a RDBS. It uses a variant of the compiled approach to hand queries to the RDBS system. The database system is extended with a secondary program to handle the large amounts of data. This paper will trace the history of the system, with particular attention to the problems which arose, and what was done to solve them. Finally, a plan for future work will be given.

### 11.8.2 Previous Approaches

The previous approaches have encountered a variety of problems. These problems include such areas as handling recursion, efficient handling of very large databases

[1]This work supported by Applied Logic Systems, Inc. under U.S. Army contract DAAB10-86-C-0551.

(of the order of gigabytes of information), and readability of programs. This section discusses some of the major attempts and the problems related to each.

### 11.8.3 The Interpreted Method

The interpreted method [Minker78] requires major changes to the underlying Prolog system. The reader is referred to [Chakravarthy] for the details of this method. The major idea is that the computation extracts the correct answer from the set of all possible solutions to each subgoal of the program. Each subgoal is seen as a restriction process. All of the possible answers from the previous state of the query are examined by the current restriction, and those not passing are removed. This method, instead of being the one-answer-at-a-time idea that Prolog adopts, provides the user with all of the answers at once.

The main problem is the amount of data that must be passed between the logic system and the RDBS, especially if they are in different processes on a single machine, or split up between two pieces of hardware. The database is going to send megabytes of information to Prolog, which will then pass it back. Much time is going to be spent in communication of this data. Moreover, buffering this much data in the two systems requires Prolog to have a database manager of its own, defeating the purpose of using the database system in the first place. Techniques exist for optimizing the data structure representing the set of solutions at each stage of the computation, but this method is felt to be inadequate for very large databases.

### 11.8.4 The Compiled Method

The compiled method [Reiter78a] postpones database queries as long as possible before sending them to the database system. A meta-interpreter could be written with definite clause grammers in Prolog to simulate this method. The interpreter would notice when a database call is being made and add it to a list of other calls that are pending. When the main program finishes running, all of these queries are sent in bulk to the database system, at which time the user gets the answer to his query back.

The problem with this method is that it is assumed that the procedures in the logic program are non-recursive. When the program is recursing, the system could possibly

pile up requests until memory was full, getting no useful work done. [Reiter78b] discusses cases where the recursion terminates, and takes advantage of this. However, not every program will have this ability. Some recursions may terminate only when an appropriate answer is retrieved from the database. One possibility is to perform a flow analysis on the program and decide when recursion will not terminate, and make the database calls earlier. However, the flow analysis may prove to be difficult.

### 11.8.5 VMProlog

VMProlog is closest to the current method used. VMProlog allows statements to be made to the SQL database system by interspersing SQL statements with Prolog goals. An evaluatable predicate SQL was added to Prolog, which allows a query to be sent to the appropriate system. An example call to SQL/DS would be

```
...,sql('select flynb,airport2 from flyex where airport1="ROME"',*1),...
```

where *1 is the variable to be instantiated to a list of the answers to the query. The query is allowed to backtrack if necessary, giving more possible instantiations of the variable.

This interface allows the database call to look like a Prolog call, but the statement of the query is not the same as if the query were stated as

```
$...,flyex(Flynb,rome,Airport2,\_,\_,\_),...$
```

which looks more like a Prolog predicate. A simple database compiler could take care of this problem.

### 11.8.6 The Syracuse Implementation

The system which follows is a testbed for trying various alternatives for Prolog/RDBS interfaces. Several evaluatable predicates were added to a version of Prolog written at Syracuse [Bowen85] to allow communication with the Ingres RDBS. Having no

large databases to test the system has been a problem. The Prolog system and the interface were written in C on a VAX780 running Unix.

A variation of the compiled approach and the VMProlog approach is used. The SQL predicate in VMProlog allows the user to keep writing in Prolog without having to delve too far into another language. The compiled method is advantageous in that it requires only slight modification to existing Prolog systems and doesn't have the problems the interpreted method has with very large databases.

### 11.8.7 The Initial Attempt

The first pass at the interface added three new predicates to Prolog: *initIngres*, *callIngres*, and *killIngres*. *initIngres* started up an Ingres sub-process, which could then be removed by *killIngres*. *initIngres* had one argument, being the database in which the predicates were to be found.

*callIngres* actually made the queries to Ingres through the EQUEL [Stonebraker76] routines supplied with Ingres. EQUEL supplies a series of C routines to allow an application programmer to call Ingres from the application program. *callIngres* had a single argument, being the predicate the user was interested in. This call was then changed into the QUEL query language for Ingres and sent to Ingres, where it was processed. The results were then returned and unified with the variables in the call. Any atoms retrieved by Ingres were installed in Prolog's name table.

For example, a call of

```
callIngres(parts(PNum,PName,pink,Weight,Qoh))
```

would cause the QUEL statements

```
range of e is parts
retrieve (e.pnum,e.pname,e.weight,e.qoh) where e.color="pink"
```

to be sent to Ingres. If no tuples were returned, *callIngres* would fail. If there were any answers, *callIngres* would unify the variables in the call with the answers returned.

If backtracking occurred, the next tuple would be retrieved, and the variables in the call would be re-bound.

This approach has several problems. First, Prolog has to know about the details of QUEL. If another database system were to be used, the evaluatable predicates would have to be rewritten. Second, Ingres returns all solution tuples at once, while Prolog can only consume one answer at a time. Finally, there is only one communication channel out of Ingres. Thus, the user is allowed only one backtrackable call to the RDBS.

### 11.8.8 DBMachine

DBMachine is an program to handle two of the problems encountered with *callIngres*: (1) the RDBS wants to retrieve all tuples answering a query at once, and (2) the need for more than one call to the database at a time, with backtracking if necessary. It is a program which allocates buffers to Prolog calls to the database, passes the call to Ingres, and stores the tuples received from Ingres in these buffers.

When Prolog needs the database system, it creates a DBMachine process, which then starts up an Ingres process with the appropriate database. Prolog requests are made to DBMachine, which gets the required information from Ingres through calls to EQUEL. Splitting up the processes is useful in seeing how to handle networks of machines talking to each other.

No QUEL statements are sent by Prolog. A much simpler request language is used by Prolog, which then can be translated to any RDBS query language, such as SQL or QUEL. This was done to increase the communication bandwidth between Prolog and the RDBS. Also, the Ingres dependence was taken away from Prolog, allowing DBMachine to call any database system without modifications to the evaluatable predicates in Prolog.

### 11.8.9 The Prolog/DBMachine Interface

Development of this interface went through two phases. The overall appearance didn't change much, but the underlying mechanisms changed.

The applications programmer talks to DBMachine through three predicates: *initDB*,

*killDB*, and *queryDB*. The first two predicates are analogous to *initIngres* and *killIngres*, except that they start and stop DBMachine.

*queryDB(Query)* passes a form of *Query* to DBMachine and instantiates any variables found in *Query* to the tuples passed back from DBMachine. *queryDB* itself is not the actual call to DBMachine, however. In order to keep the routines in C from becoming unmanageable, and to allow Prolog to backtrack over the stream of tuples that DBMachine has generated in response to the query, *queryDB* is written in Prolog as follows:

```
queryDB(Query) :-
   requestDB(Query,BufferID),
   getAnswers(g(BufferID),Query).

getAnswers(ID,Query) :-
   answerDB(ID,Query).
getAnswers(g(BufferID),_) :-
   BufferID < 0,
   !,fail.
getAnswers(ID,Query) :-
   getAnswers(ID,Query).
```

Notice that the two predicates called *requestDB* and *answerDB* are hidden from the casual user because they have a procedural flavor, whereas the top level call *queryDB* does not. *requestDB* is called with two arguments, one instantiated to the query to be made, and the second, a variable to be instantiated to the number (a positive integer) of the buffer in DBMachine where the results will reside. The results are then retrieved from the buffer by *answerDB*, which is encapsulated in *getAnswers* to allow Prolog to backtracking over the query. The internals of *queryDB* are hidden from the user so that *answerDB* can make a destructive assignment to signal when no more answers are to be found in the buffer. If there are tuples left in the buffer, *answerDB* will set the variables in *Query* to their proper values for the next tuple in the buffer. If the program requires more answers for *Query*, the first and second clauses will fail, and *getAnswers* will be called again. The combination of the first and third clauses for *getAnswers* effects the iteration through the buffer as Prolog backtracks over *Query*. If the buffer is empty, *answerDB* changes the *BufferID* to -1 and fails. This is caught by the second clause of *getAnswers*, which causes *getAnswers* to fail. Any atoms retrieved through *answerDB* are placed in Prolog's name table.

*requestDB* modifies *Query* in order to limit the amount of information sent to DBMachine. In the first phase of the interface development, this information consisted of the predicate name, a list of numbers describing which columns contained the uninstantiated variables, and a list of column numbers corresponding to instantiated values, along with those values. *answerDB* would examine the Prolog structure of *Query*, noticing where the variables were, and get the corresponding tuple values back from the buffer in DBMachine.

### 11.8.10 Problems with Phase One and Their Solution

Phase one of this interface only allowed the user to hand one call to DBMachine at a time. To solve goals of the form

```
...,a(A,B,C),b(F,A,R),...
```

where *a* and *b* were database calls, the user had to write them as

```
..., queryDB(a(A,B,C)), queryDB(b(F,A,R)), ...
```

This had the potential of retrieving the same information from *b* multiple times as multiple *a* tuples were extracted before a pair of solutions with a common *A* were found. However, this operation is just a join in standard RDBS terminology, and joins are something RBDSs do well.

During the second phase of interface development, a database compiler was written to allow the users to write programs with little thought about database systems. The compiler attempts to retrieve database information as efficiently as possible. The user places *db* declarations at the beginning of the application program, stating which predicates are database calls. For example, the user could say

```
:- db(parts/5), db(item/6)
```

to declare the two 5- and 6-place predicates *parts* and *item* as residing in the database. The program is then read in using *dbConsult(File)*, which reads *File* and asserts

rewritten forms of the program clauses. This rewriting has two phases. The first simply scans the clause and replaces all calls having predicates declared with *db* by a *queryDB* applied to the call. Then, a pass over the rewritten clause optimizes the call. Currently, the only optimization attempted is that contiguous *queryDB*'s are merged into one *queryDB* call. For example,

```
a :- parts(A,B,C),item(A,F), F < 12000.
```

is changed to

```
a :- queryDB([parts(A,B,C),item(A,F)]), F < 12000.
```

At the moment, any other possible optimizations, such as passing range checks as shown above, are not performed. Also, disjunctions are ignored. As more optimizations are noticed, they will be added.

In the first phase of interface development, if a call such as *a(A,A)* was made, two items were passed back from DBMachine and unified together. This was fixed in the second phase. After sending information on each separate predicate to DBMachine, *requestDB* notices where variables are repeated and sends this information along. The restricted QUEL expression generated by the above *queryDB* would be

```
range of e0 is parts
range of e1 is item

retrieve (e0.col1,e0.col2,e0.col3,e1.col2)
    where e0.col1 = e1.col1
```

which is processed more quickly by Ingres than the unrestricted QUEL

```
range of e0 is parts
range of e1 is item
retrieve (e0.col1,e0.col2,e0.col3,e1.col1,e1.col2)
```

which has many more possible solutions. In one test, the query was a join on the first argument of parts and item, where item was a 6 place predicate with 20 tuples, and parts was a 5 place predicate with 14 tuples. Without the restriction, the cartesian product of 280 tuples was retrieved by Ingres, taking around 15 seconds. Approximately 10 seconds were then used to unify the possible combinations together until the correct tuple was found. With the restriction added, less than a second of processing time was needed. Variables with multiple occurrences in the query only elicit one value to be returned from DBMachine. For example, the above restricted join would return four items per tuple back to Prolog.

### 11.8.11 Future Research

Many problems remain to be solved. One problem is clogging of the atom table. When strings are returned from DBMachine, they are stored in Prolog's name table. With a large database, the name table will soon be clogged with new atoms from the database, even though many may be useless to the program. One possible solution to this problem is to have DBMachine assign a unique identifier to each atom Prolog hasn't already entered in the atom table. Also, more optimizations on the query can be performed, such as the range check mentioned above. Allowing variables and structures to reside in the database would be helpful. Another modification to improve efficiency would be to have DBMachine return Warren Abstract Machine code [Warren83]. This could then be executed by Prolog to retrieve the possible answers to the query.

The problems above only relate to retrieval from the database. In actual applications, the Prolog system will have to make updates to the database. An approach similar to [Warren84], with appropriate optimizations for bulk updates and the like, will be implemented.

### 11.8.12 Conclusion

Relational database systems and logic are so closely related, it seems that they must be joined to solve the problems of large-scale knowledge base management. There has been much debate as to the right approach to amalgamate the two. The interface system constructed in this project provides a flexible testbed in which to explore solutions to the difficult problems arising in this amalgamation. Work with this

interface has indicated solutions to some questions such as the join problem. It is felt that the remaining problems can be similarly solved, resulting in a powerful amalgam of Prolog and relational database systems.

### 11.8.13 References


Bowen, K.A., Buettner, K.A., Cicekli, I., and Turk, A., [1985]: *The Design and Implementation of a High-Speed Incremental Portable Prolog Compiler*, Tech Report CIS-85-4, Syracuse University. To appear in The International Logic Programming Conference '86 Proceedings.

Chakravarthy, U.S., Minker, J., and Tran, D.: *Interfacing Predicate Logic Programs and Relational Databases*, University of Maryland, unpublished draft.

Gallaire, H., and Minker, J. [1978]: Logic and Databases, Plenum Press, New York.

Minker, J., [1978]: *An Experimental Data Base System Based on Logic.* In: Logic and Databases (eds. Gallaire, H. and Minker, J.), Plenum Press, New York, pp. 107-148.

Reiter, R., [1978a]: *Deductive Question Answering on Relational Data Bases.* In: Logic and Databases (eds. Gallaire, H. and Minker, J.), Plenum Press, New York, pp. 149-176.

Reiter, R., [1978b]: *Structuring a First-order Database*, Proceedings of the Canadian Society for Computational Studies of Intelligence.

Stonebraker, M., Wong, E., Kreps, P., and Held, G. [1976]: *The Design and Implementation of Ingres*, ACM Transactions on Database Systems, 1:3, (September 1976).

Warren, D.H.D., [1983]: *An Abstract Prolog Instruction Set*, SRI Technical Report.

Warren, D.S., [1984]: *Database Updates in Pure Prolog*, Proceedings of the International Conference on Fifth Generation Computer Systems.

## 11.9 Hyung-Sik Park[2]

*Negation and Databases*

Hyung-Sik Park was a visiting assistant professor of Computer and Information Science during the academic year 1986-87, and worked on the grant during the spring term of 1988. (He accepted a position at the University of Iowa in June of 1988.) His research area is the interaction between logic deduction from databases and the assumption of the Generalized Closed World Assumption (GCWA) for those databases, which was the subject of his dissertation at Northwestern University under the direction of Prof. Lawrence J. Henschen. Under this approach, one separates the complete database available to the program into two distinct parts: The Extensional Database (EDB) consisting of ground atoms (facts) and the Intensional Database (IDB) consisting of all other available clauses. The assumption is that in large applications the size of the EDB will dwarf that of the IDB, and that typically, the EDB will be maintained on secondary storage while the IDB will often reside in main memory. The concern is that the solution of complex queries will lead to large volumes of retrievals of facts from the EDB. Since retrieval from the EDB will be measured in milliseconds as opposed to microsecond retrievals from the IDB, this would lead to serious inefficiencies in applications.

The approach taken here to this problem is based on the general compilation philosophy followed in the rest of the project: Determine at compile-time the EDB retrievals which can follow from use of a member of the IDB. Then at run time, various optimizations to speed availability of the EDB facts can be applied. This can range from semi-symbolic execution of the program – batching all retrievals to the end of symbolic execution, followed by retrieval and final resolution of the solution – to initiating parallel retrieval and caching of the EDB facts associated with an IDB rule the moment it is evident at run-time that the rule will be executed. For Horn clause databases, the basic theory of this approach has been worked out in [Chang,1981], [Henschen and Naqvi, 1984], and [Reiter, 1978a].

The treatment of negative information causes an increase in problems. Because of the potentially large volume of negative facts which must be stored if explicit representation were to be used, it is preferrable to represent negative facts implicitly. This leads to the Closed World Assumption (CWA - [Reiter,1978b]): A ground fact is assumed to be false (i.e., its negation is true) if it cannot be deduced from the combina-

[2]This section written by K.A. Bowen.

tion of the EDB and IDB. Otherwise stated, this is the *principle of negation by failure:* A negated ground atom is provable if the attempt to prove (via a complete positive deduction procedure) the unnegated ground atom fails. For Horn databases, the relation between negation by failure and logical negation is well-understood ([Clark, 1978]). However, for non-Horn IDBs, the CWA leads to contradictions. For example, if the IDB consists of (p v q) alone, then neither p nor q is a logical consequence of the DB, so that under the CWA, both -p and -q would be provable, a contradiction. The difficulty arises because p (and also q) is *indefinite* with respect to the DB: neither p nor q is a logical consequence of the DB. The example demonstrates that the negation by failure approach does not distinguish between genuinely false atoms (relative to the DB) and those which are indefinite relative to the DB.[3]

Define PIGC to be the set of minimal positive ground clauses implied by the DB, where a clause is minimal if it is not properly subsumed by any positive clause deducible from the DB. For a ground atom q, PIGC[q] consists of those elements of PIGC in which q occurs positively as a subformula. The Generalized Closed World Assumption states tnat if q is a ground atom, then -q can be assumed true if q is not deducible from DB *and* q is not indefinite with respect to DB. Let us write GCWA(DB,-q) for this state of affairs. It follows from [Minker, 1982] that GCWA(DB, -q) if and only if PIGC[q] is empty. It follows that the problem of coping with indefinite formulae can be reduced from treating the entire set of indefinite formulae (with respect to DB) can be reduced to computing the indefinite formulae relevant to the query at hand.

Further reductions are possible. The following representations are known (Henschen and Park, [Henschen and Park, 1986]:

(1) PIGC[q] with respect to DB is equivalent to PIGC[q] with respect to $EDB \cup NUGF \cup NH[q] \cup PSUB[nhi]$, where

$CDB = IDB \cup NNUC,$

[3]The unpleasant nature of DBs which leave some formulas indefinite is long-established: Classical logic and model theory – e.g., modern proofs of the Completeness Theorem – exensively utilize methods (Lindenbaum's Lemma) which embed initial consistent theories or DBs in complete extensions; i.e. in supersets which leave no formulas indefinite. Unfortunately, although Lindenbaum's Lemma guarantees the existence of complete extensions (assuming the Axiom of Choice), the problem of obtaining such extensions is recursively unsolvable. Consequently, even in those settings where passing to a complete extension would be logically reasonable, it is not computationally possible. Hence the need to compute the set of indefinite formulas of the original theory. -*KAB*

(2) PIGC[q] with respect to DB is equivalent to PIGC[q] with respect to $EDB \cup NH[q] \cup PSUB[nhi]$, where

$$CDB = IDB \cup NC,$$

In both cases, NH[q] is the set of minimal non-Horn clauses containing a positive occurrence of the predicate of q and are derivable from CDB, NNUC is the set of negative nonunit clauses, and NUGF is the set of negative unit ground facts, NC is the set of negative clauses, and PSUB[nhi] is the set of clauses derivable from CDB and which potentially subsume some clause in NH[q].

All of the foregoing results are valid for DBs whose formulae contain no function symbols. During the spring term, Park investigated methods for possible extension of the results to settings in which function may be present, as well as possible further improvements of the reductions and resulting computations of GCWA[q,DB]. Several suggestive special cases appeared, but as yet no general conclusions can be drawn.

Park also organized and conducted a research seminar on Expert Database Systems for the staff of the grant, as well as other graduate students in the department. The outline of the topics of the seminar was:

- Introduction of expert systems, databases, and expert database systems.
- Knowledge-based systems, knowledge representation, logical analysis of knowledbases, incompleteness, commonsense reasoning, non-monotonicity, and reason-maintenance systems.
- Database management systems, semantic data modelling, database constraints, dependencies, and normal forms, dextensions of DBMSs, including deductive databases, incomplete databases, and temporal databases.
- Knowledge base management systems and architectures, log c-based data languages, recursion, complex objects, object-oriented paradigms in KBMSs, constraint management, semantic query optimization, knowledge engineering in DBMSs, intelligent KB-interfaces.

## References

[Chang] Chang,C.L., *On evaluation of queries containing derived relations*, in **Advances in Database Theory**, vol. 1, H.Gallaire, J.Minker, and J.M. Nicolas, eds,

Plenum Press, New York, 1981, 235-260.

[Clark] Clark, K.L., *Negation as failure*, in **Logic and Databases**, H.Gallaire and J.Minker, eds, Plenum Press, New York, 1978, 293-324.

[Henschen and Naqvi], Henschen, L.J. and Naqvi, S., *On compiling queries in recursive first-order databases*, **JACM**, 31:1(1984),47-85.

[Henschen and Park] Henschen, L. and Park, H-S., *Indefinite and GCWA inference in indefinite deductive databases,* in **Proc. AAAI National Conference**, 1986.

[Minker] Minker,J., *On indefinite databases and the closed world assumption*, in **Lecture Notes in Computer Science**, 138, Springer Verlag, 1982, 292-308.

[Reiter,1978a] Reiter,R., *Deductive question answering on relational databases*, in **Logic and Databases**, H.Gallaire and J.Minker, eds, Plenum Press, New York, 1978, 149-177.

[Reiter,1978b] Reiter,R., *On closed world databases*, in **Logic and Databases**, H.Gallaire and J.Minker, eds, Plenum Press, New York, 1978, 55-76.

## 11.10 V. S. Subrahmanian

*Theory of Logic Programming*

My work on the project concentrated on the development of a mathematical basis for classical and non-classical logic programming. In particular, I developed, jointly with Aida Batarekh, a topological theory of logic programming model theory, while both alone and/or jointly with A. N. Hirani, I developed an algebraic basis for logic programming. I also concentrated on the study of several different non-classical logic programming languages.

### 11.10.1 Logic Programming with Non-Classical Logics.

I have been involved in the development of a family of non-classical logic programming languages that can be semantically characterized in terms of fixed-point theory. Proposals for logic programming with specific logics (e.g. quantitative logics, paraconsistent logics, etc.) were later generalized to yield a generalized declarative semantics for logic programming over certain kinds of partially ordered sets of truth values. This declarative semantics is independent (to some extent) of the syntactic nature of a non-classical logic program. In addition, I developed a proof-theoretic generalization of SLD-resolution that is sound and complete for many-valued logic programs (whose set of truth values is a complete lattice).

### 11.10.2 Paraconsistent Reasoning.

The design of very large knowledge bases may sometimes result in some inaccuracies. Paraconsistent logics provide a framework for reasoning in the presence of inconsistency (in the sense of classical logic) via non-classical model theory. Howard Blair and I have worked on a formal theoretical framework for mechanical reasoning in the presence of inconsistency. More recently, M. Chakrabarti and I are working on the semantics of general logic programs (even those whose completions are inconsistent) with a view to developing a theory of local and global consistency. Newton da Costa and I are investigating syntactic consequence relations that lead to paraconsistent logics with a view to developing a proof-theoretic characterization of inconsistent databases.

### 11.10.3 Topological Methods in Logic Programming.

Aida Batarekh and I studied the topological properties of the space of models of logic programs (and also arbitrary sentences in first order logic). We then derived results on the fixed-points of non-monotonic operators that map structures to structures. As a consequence of some results on the (topological) continuity of the well-known operator $T_P$ associated with a logic program $P$, we were able to obtain necessary and sufficient conditions on the consistency of $comp(P)$ (when $P$ is either a function free or covered logic program).

### 11.10.4 Metalogic Programming.

My paper *Foundations of Metalogic Programming* is the first paper to address the problem of developing a formal theoretical framework for reasoning about the amalgamation of object language and metalanguage in logic programming. It is a companion to the paper by Pat Hill and John Lloyd that considers metalevel programming *without* the amalgamation.

### 11.10.5 Types in Prolog.

Lee Naish and I have jointly developed a framework for incorporating types in Prolog. For programming purposes, our view is that type declarations are useful, and our semantics essentially characterizes logic programming augmented with type declarations.

### 11.10.6 Auto-Epistemic Logics.

Wiktor Marek and I are currently studying the connections between differing treatments of negation in logic programming and AI. In addition, we have studied the complexity of determining the truth of a formula in a stable expansion of an auto-epistemic first order theory.

### 11.10.7 Nuclear Systems.

A nuclear system is essentially a triple $(S, \vdash, Q)$ where $S$ is a non-empty set, $Q$ is the set of existential queries that can be expressed in some fixed but arbitrary first order language, and $\vdash$ is a binary relation between $S$ and $Q$. For example, $S$ may be a set of theories, and $\vdash$ may be an entailment relation, or $S$ may be a set of interpretations for a first order language and $\vdash$ may be a model-theoretic satisfaction relation, or $S$ may be a set of theories and $\vdash$ may be a non-monotonic forcing relation. When the nuclear system satisfies some simple conditions, $S$ turns out to be a compact Hausdorff space (under a topology induced by the $\vdash$ relation). One can now study the fixed-points of non-monotonic closure operators in terms of topological results.

### 11.10.8 Algebraic Theory of Logic Program Construction.

Given a logic program $P$, the operator $T_P$ associated with $P$ is closely related to the intended meaning of $P$. Given a first order language $L$ that is generated by finitely many non-logical symbols, our aim is to study the algebraic properties of the set $\{T_P \mid P$ is a general logic program in language $L\}$ with certain operators on it. For the operators defined in this paper the resulting algebraic structure is a bounded distributive lattice. Our study extends (to the case of general logic programs), the work of Mancarella and Pedreschi who initiated a study of the algebraic properties of the space of pure logic programs. We study the algebraic properties of this set and identify the ideals and zero divisors. In addition, we prove that our algebra satisfies various *non-extensibility* conditions. This algebraic study shows promise of leading to a theory of modules in logic programming.

### 11.10.9 Protected Completions of Logic Programs.

The notion of protected completion $pc(P)$ of a logic program $P$ was introduced by Jack Minker and Don Perlis. The Minker-Perlis proposal laid the foundation for reasoning via protected completions for pure, function free logic programs. We extend their work by characterizing protected completions of general logic programs. Thus, both restrictions in the Minker Perlis proposal are removed. Operational algorithms are also developed. This work is being carried on jointly with James Lu.

### 11.10.10 Theorem Proving in Systems with Equality.

James Lu and I studied certain open problems concerning the soundness and completeness of various problems in RUE–NRF deduction. We proved, amongst other results, that RUE–NRF deduction in stron form is incomplete contradicting existing published results of V. Digricoli and M. Harrison. Our disproof has since been acknowledged as being correct by V. Digricoli. Since then, we have worked on the problem of termination of the viability check in RUE–NRF deduction using a method based on AND/OR graphs.

### 11.10.11 Accepted/Published Papers

**Dissertation:** "Computational Reasoning with Non-Classical and Paraconsistent Logics." Advisor: Howard A. Blair. Ph.D., August, 1989, School of Computer and Information Science, Syracuse University, Syracuse, NY 13244.

1. Protected Completions of First Order General Logic Programs, accepted for publication in: *Journal of Automated Reasoning.* (with James Lu). Sep. 1988.

2. Topological Model Set Deformations in Logic Programming, accepted for publication in: *Fundamenta Informaticae*, North Holland. (with A. Batarekh).

3. A Ring-Theoretic Basis for Logic Programming, accepted for publication in: *International Journal of Foundations of Computer Science.*

4. Paraconsistent Logic Programming, *7th Foundations of Software Technology & Theoretical Computer Science Conf.*, Lecture Notes in Computer Science, Vol. 287, pps 340–360, Springer-Verlag. An extended version of this paper has been accepted for publication in *Theoretical Computer Science.* (with Howard Blair).

5. AND-OR Graphs Applied to RUE-Resolution, accepted for publication in: *Proc. 11th International Joint Conference on Artificial Intelligence*, Detroit, Michigan, Aug. 1989. (with V.J. Digricoli and J. J. Lu).

6. Paraconsistent Foundations for Logic Programming, accepted for publication in *J. of Non-Classical Logic*, (with Howard Blair).

7. Algebraic Foundations of Logic Programming, I: The Distributive Lattice of Logic Programs, accepted for publication in *Fundamenta Informatica*, North Holland. (with A. N. Hirani).

8. Mechanical Proof Procedures for Many-Valued Lattice-Based Logic Programming, accepted for publication in: *Journal of Non-Classical Logic.*

9. The Relationship Between Logic Program Semantics and Non-Monotonic Reasoning, accepted for publication in: *Proc. 6th International Conference on Logic Programming*, (eds. G. Levi and M. Martelli), pps 600–617, Lisbon, Portugal, June 1989, MIT Press. (with Wiktor Marek).

10. On the Expressive Power of Annotation Based Logic Programs, accepted for publication in: *Proc. 1989 North American Conference on Logic Programming*, (eds. E. Lusk and R. Overbeek), Cleveland, Ohio, Oct. 1989, MIT Press. (with Michael Kifer).

11. Th[illegible]ery Topology in Logic Programming, in: *Proc. International Symposium on [illegible]oretical Aspects of Computer Science*, Lecture Notes in Computer Science Vol. 349, pps 375–387, Springer Verlag. (with Aida Batarekh).

12. Query Processing in Quantitative Logic Programming, *Proc. 9th Conference on Automated Deduction*, Lecture Notes in Computer Science Vol. 310, pps 181-200, Springer, (eds. E. Lusk and R. Overbeek). May 1988.

13. QUANTLOG: A System for Approximate Reasoning in Inconsistent Formal Systems, *Proc. 9th Conference on Automated Deduction*, Lecture Notes in Computer Science Vol. 310, pps 746-747, Springer–Verlag, (eds. E. Lusk and R. Overbeek). (with Z. Umrigar). System Summary. May 1988.

14. Foundations of Metalogic Programming, *Proc. of the Workshop on Meta-Programming in Logic Programming*, (ed. John Lloyd), pps 53-66, Bristol, England, June 1988. An extended version of this paper is to be published this summer in a book edited by H. Abramson and M. Rogers. The book is to be published by MIT Press.

15. Semantical Equivalences of (Non-Classical) Logic Programs, in: *Proc. 5th International Conference/ Symposium on Logic Programming*, eds. R. Kowalski and K. Bowen, pps 960–977, MIT Press. (with A. Batarekh).

16. Intuitive Semantics for Quantitative Rule Sets, in: *Proc. 5th International Conference/Symposium on Logic Programming*, eds. R. Kowalski and K. Bowen, pps 1036–1053, MIT Press, August 1988.

17. On the Semantics of Quantitative Logic Programs, *Proc. 4th IEEE Symp. on Logic Programming*, pps 173-182, Computer Society Press. Sep. 1987.

18. FLOG: A Logic Programming System Based on a Six-Valued Logic, *AAAI/Xerox Second Intl. Symp. on Knowledge Engg.*, Madrid, Spain, (with R.Anand). April 1987.

**Submitted Papers**

19. Completeness Issues in RUE–NRF Deduction, submitted to the *Journal of the ACM* Currently being revised in accordance with referees' comments. (with James Lu).

20. Algebraic Foundations of Logic Programming, II: The Space of Multivalued and Paraconsistent Logic Programs, submitted to *Acta Informatica.* Feb. 1989.

21. Approximate Reasoning in Logic Programming, submitted to *New Generation Computing.* March 1988.

22. Strong Completeness Results for Paraconsistent Logic Programming, currently being revised for publication in a special volume on Theorem Proving in Non-Classical Logics, ed. Michael McRobbie, (with Howard Blair). Aug. 1988; revised, Augu. 1989.

23. The Relationship Between Stable, Supported, Default and Auto-Epistemic Semantics for General Logic Programs, submitted to *Theoretical Computer Science.* (with Wiktor Marek). Jan. 1989.

24. Paraconsistent Logics as a Formalism for Reasoning About Inconsistent Knowledge Bases, submitted to: *Journal of Artificial Intelligence in Medicine.* (with Newton C. A. da Costa).

25. Y-Logic: A Framework for Reasoning about Chameleonic Programs with Inconsistent Completions, submitted to: *Fundamenta Informaticae*, April 1989.

26. A Petri Net Model for Reasoning in the Presence of Inconsistency, submitted to: *IEEE Transactions on Data and Knowledge Engineering*, June 1989, (with T. Murata and T. Wakayama).

27. Existential, Null and Partial Values in Disjunctive Deductive Databases, submitted to: *Journal of Automated Reasoning*, June 1989.

28. The Viability Check in Equality Based Binary Resolution, to be submitted. (with James J. Lu).

29. Paraconsistent Disjunctive Deductive Databases, submitted to: *Theoretical Computer Science*, July 1989.

30. The Paraconsistent Logics $\mathbf{P}\mathcal{T}$, to be submitted. (with N.C.A. da Costa and C. Vago).

## 11.11 Supported Students

### Hamid Bacha

PhD, School of Computer & Information Science

Syracuse University, Syracuse, NY, December, 1989

*Dissertation:* METAPROLOG: Design, Implementation, and Application to a Medical Expert System in Acid-Base and Electrolyte Disorders.

Advisor: Kenneth A. Bowen

### Aida Batarekh

PhD, School of Computer & Information Science

Syracuse University, Syracuse, NY, August, 1989

*Dissertation:* Topological Aspects of Logic Programming.

Advisor: Kenneth A. Bowen

### Kevin Buettner

MS, School of Computer & Information Science

Syracuse University, Syracuse, NY, December, 1986.

### Ilyas Cicekli

PhD expected, School of Computer & Information Science

Syracuse University, Syracuse, NY, expected January, 1990.

### Keith Hughes

MS, School of Computer & Information Science

Syracuse University, Syracuse, NY, May, 1989.

*Thesis:* Prolog as a Uniform Interface to a Heterogeneous Distributed Database.

**Andrew Turk**

BS. School of Computer & Information Science

Syracuse University, Syracuse, NY, December, 1986

**V.S. Subrahmanian**

PhD, School of Computer & Information Science

Syracuse University, Syracuse, NY, August, 1989

em Dissertation: Computational Reasoning with Non-Classical and Paraconsistent Logics.

Advisor: Howard A. Blair.

**Toshiro Wakayama**

PhD, School of Computer & Information Science

Syracuse University, Syracuse, NY, August, 1989

*Dissertation:* Reasoning with Indefinite Information in Resolution-based Languages.

Advisor: Howard A. Blair.

## 11.12 Publications


1. R. Anand and V. S. Subrahmanian. *FLOG: A Logic Programming System Based on a Six-Valued Logic*, **AAAI/Xerox Second Intl. Symp. on Knowledge Eng.**, Madrid, Spain, April 1987.

2. Apt, K. R. & Blair, H. A. *Arithmetic Classification of Perfect Models of Stratified Programs*, (preliminary version) Jan. 1988. **The Proceedings of the Joint Fifth International Logic Programming Conference and Fifth IEEE Symposium on Logic Programming** Seattle, Washington, August, 1988. Also: Syracuse University Logic Programming Research Group Technical Report LPRG-TR88-11.

3. Apt, K. R., Blair, H. A., & Walker, A. *Towards a Theory of Declarative Knowledge*, in **Foundations of Deductive Databases and Logic Programming,** Jack Minker, ed. Morgan-Kaufmann, Los Altos, CA. 1988. pp. 89-148.

4. Apt, K. R. & Blair, H. A. *Recursion-free Programs.* Syracuse University Logic Programming Research Group Technical Report LPRG-TR-88-12.

5. Apt, K. R. & Blair, H. A. *Arithmetic Classification of Perfect Models of Stratified Programs.* Invited Submission to **Fundamenta Informatica.** (To appear.)

6. H. Bacha. *Meta-level Programming: A Compiled Approach*, **Proceedings of the Fourth International Conference on Logic Programming.** Melbourne, Australia, 1987. J-L. Lassez, (ed.)

7. H. Bacha. *MetaProlog Design and Implementation*, **Proceedings of the Fifth International Conference on Logic Programming**, Seattle, Wa. 1988. K.A. Bowen and R. Kowalski (eds.)

8. H. Bacha. *A Prototype Medical Expert System in Acid-Base Disorders Implemented in MetaProlog* **Journal of the AIST**, 1(3), Spring, 1989.

9. H. Bacha, K. Bowen and C. Carvounis. *Clinical vs. Pathophysiological Knowledge in Medical Expert Systems*. Preliminary version to appear in the **Proceedings of the AIST Conference**, October, 1989.

10. A. Batarekh and V. S. Subrahmanian. *Semantical Equivalences of (Non-Classical)*

*Logic Programs*, **Proc. 5th International Conference on Logic Programming**, Seattle, MIT Press, pp 960-977, 1988.

11. A. Batarekh and V.S. Subrahmanian. *A $T_4$ Space of Models of Logic Programs and their completions, I: Foundations*, Technical Report, Logic Programming Research Group, LPRG-TR-88-15.

12. A. Batarekh and V.S. Subrahmanian. *The Query Topology in Logic Programming*, **Proc. Intl. Symp. on Theoretical Aspects of Computer Science**, Lecture Notes in Computer Science, v. 349 Springer Verlag, pp. 375-387, Feb. 1989.

13. A. Batarekh and V.S. Subrahmanian. *Topological Model Set Deformations in Logic Programming*, to appear in: **Fundamenta Informatica.**

14. H. A. Blair. *Decidability in the Herbrand Base*, **Workshop on Deductive Databases and Logic Programming**, Washington D.C. Aug 18-22, 1986. Revised as Syracuse University Logic Programming Research Group Technical Report LPRG-TR88-13.

15. H. A. Blair. *Canonical Conservative Extensions of Logic Program Completions*, **IEEE Symposium on Logic Programming**, San Francisco, August, 1987. pp. 154-161.

16. H. A. Blair. *Metalogic Programming and Direct Universal Computability*, To appear in the **Proceedings of the Workshop on Metalogic Programming, 1988**, H. Abramson and M. Rogers (eds.), MIT Press. Syracuse University Logic Programming Research Group Technical Report LPRG-TR88-23.

17. H. A. Blair and V.S. Subrahmanian. *Paraconsistent Logic Programming*, **Proc. 7th Conference on Foundations of Software Technology and Theoretical Computer Science**, Lecture Notes in Computer Science, Vol. 287, pps 340-360, Springer Verlag. Extended version to appear in: **Theoretical Computer Science.**

18. H. A. Blair and V. S. Subrahmanian. *Paraconsistent Foundations for Logic Programming*, to appear in: **Journal of Non-Classical Logic.**

19. H. A. Blair, A. L. Brown, and V. S. Subrahmanian. *A Logic Programming Semantics Scheme, Part I.* Jan. 1988. Syracuse University Logic Programming

Research Group Technical Report LPRG-TR88-8.

20. K. A. Bowen. *Meta-level programming and knowledge representation*, **New Generation Computing,** v.3(1985), pp.359-383.

21. K. A. Bowen. *New Directions in Logic Programming*, (invited address), **Proc. 1986 ACM Annual Computer Science Conference**, Cincinnati, pp. 19-27.

22. K. A. Bowen. *Meta-Level Techniques in Logic Programming*, (invited 1-hour talk) **Proceedings of the Artificial Intelligence '86 Conference**, Singapore, March, 1986.

23. K. A. Bowen. *On the Use of Logic: Reflections on McDermott's Critique of Pure Reason*, invited paper for a special issue of **Computational Intelligence**, Vol. 3, 1987, pp. 165-168.

24. K. A. Bowen and R. A. Kowalski. Editor, **Logic Programming: Proceedings of the Fifth International Conference and Symposium**, (Seattle, 1988), Cambridge, MA: MIT Press, 1,690pp.

25. K. A. Bowen, K. Buettner, I. Cicekli & A. Turk. *The Design and Implementation of a High-speed Incremental Portable Prolog Compiler*, **Proceedings of the 1986 International Logic Programming Conference**, London, July, 1986, pp. 650-656.

26. K. Buettner, *Fast Decompilation of Compiled Prolog Clauses,* **Proc. 3rd Int'l Logic Programming Conf.**, London, 1986, pp. 662-668.

27. I. Cicekli. *An Abstract MetaProlog Engine for MetaProlog*, in: **Proc. of the Workshop on Meta-Programming in Logic Programming,** Bristol, England, June 1988. To appear in the **Proceedings of the Workshop on Metalogic Programming, 1988**, H. Abramson and M. Rogers (eds.), MIT Press.

28. V. J. Digricoli, J. J. Lu, and V. S. Subrahmanian. *AND-OR Graphs Applied to RUE-Resolution*, accepted for publication in: **Proc. 11th International Joint Conference on Artificial Intelligence**, Detroit, Michigan, Aug. 1989.

29. L. Henschen and H-S. Park. *Indefinite and GCWA Inference in Indefinite Deductive Databases*, in **Proc. AAAI National Conference**, 1986.

30. A. N. Hirani and V. S. Subrahmanian. *Algebraic Foundations for Logic Programming, I: The Distributive Lattice of Logic Programs*, to appear in: **Fundamenta Informatica.**

31. M. Kifer and V. S. Subrahmanian. *On the Expressive Power of Annotation Based Logic Programs*, To appear in: **Proc. 1989 North American Conference on Logic Programming**, E. Lusk and R. Overbeek (eds.), Cleveland, Ohio, Oct. 1989, MIT Press.

32. J. Lu and V. S. Subrahmanian. *Protected Completions of First Order General Logic Programs*, To appear in: **Journal of Automated Reasoning**, Sept. 1988.

33. W. Marek and V. S. Subrahmanian. *The Relationship Between Logic Program Semantics and Non-Monotonic Reasoning*, To appear in: **Proc. 6th International Conference on Logic Programming**, G. Levi and M. Martelli (eds.), pps 600–617, Lisbon, Portugal, June 1989, MIT Press.

34. Payne, T. H. & Wakayama, T. *Case Inference in Resolution-Based Languages*, **Proc. 9th Conference on Automated Deduction**, Lecture Notes in Computer Science, Springer-Verlag, May 1988.

35. V. S. Subrahmanian. *On the Semantics of Quantitative Logic Programs*, **Proc. 4th IEEE Symp. on Logic Programming**, pps 173-182, Computer Society Press. Sept. 1987.

36. V. S. Subrahmanian. *Foundations of Metalogic Programming*, **Proc. of the Workshop on Meta-Programming in Logic Programming**, J. Lloyd (ed.), pps 53–66, Bristol, England, June 1988. Extended version to appear in: **Proceedings of the Workshop on Metalogic Programming, 1988**, H. Abramson and M. Rogers (eds.), MIT Press.

37. V. S. Subrahmanian. *Intuitive Semantics for Quantitative Rule Sets*, **Proc. 5th International Conference/Symposium on Logic Programming**, K. Bowen and R. Kowalski (eds.), MIT Press, Aug. 1988.

38. V. S. Subrahmanian. *Query Processing in Quantitative Logic Programming*, **Proc. 9th International Conference on Automated Deduction**, E. Lusk and R. Overbeek (eds.), Lecture Notes in Computer Science Vol. 310, pps 81–100, Springer Verlag, 1989.

39. V. S. Subrahmanian. *Mechanical Proof Procedures for Many Valued Lattice Based Logic Programming*, To appear: **Journal of Non-Classical Logic.**

40. V. S. Subrahmanian. *A Ring-Theoretic Basis for Logic Programming*, To appear: **International Journal of Foundations of Computer Science.**

41. V. S. Subrahmanian and Z. Umrigar. *QUANTLOG: A System for Approximate Reasoning in Inconsistent Formal Systems*, (System Summary) **Proc. 9th Conference on Automated Deduction**, E. Lusk and R. OVerbeek (eds.) Lecture Notes in Computer Science, vol. 310, pps 746–747, Springer-Verlag, May 1988.

42. A. Turk. *Compiler optimizations for the WAM,* **Proc. 3rd Int'l Logic Programming Conf.**, London, 1986, pp. 656-662.

### 11.12.1 Papers in Submission and to be Submitted

1. H. Bacha. *Beyond the WAM: A PAM for the CAM. (A Prolog Abstract Machine for Content-Addressable Memory.)*, **Workshop on Prolog and Computer Architecture**, October, 1989. (To be submitted to a technical journal.)

2. H. Bacha and S. Khanna. *Program Verification Using Meta-Level Logic Programming*, (To be submitted to a technical journal.)

3. H. A. Blair and V. S. Subrahmanian. *Strong Completeness Results for Paraconsistent Logic Programming*, submitted to a technical journal.

4. H. A. Blair and V. S. Subrahmanian. *Strong Completeness Results for Paraconsistent Logic Programming.* Currently being revised for publication in a special volume on *Theorem Proving* in **Non-Classical Logics**, Michael McRobbie (ed.), Aug. 1988; revised, Aug. 1989.

5. N. C. A. da Costa and V. S. Subrahmanian. *Paraconsistent Logics as a Formalism for Reasoning About Inconsistent Knowledge Bases*, submitted to: **Journal of Artificial Intelligence in Medicine.**

6. N. C. A. da Costa, V. S. Subrahmanian and C. Vago. *The Paraconsistent Logics PT*. To be submitted.

7. J. Lu and V. S. Subrahmanian. *Completeness Issues in RUE-NRF Deduction*, submitted to the **Journal of the ACM**. Currently being revised in accordance with referees' comments.

8. J. Lu and V. S. Subrahmanian. *The Viability Check in Equality Based Binary Resolution*. To be submitted.

9. W. Marek and V. S. Subrahmanian. *The Relationship Between Stable, Supported, Default and Auto-Epistemic Semantics for General Logic Programs*. Submitted to **Theoretical Computer Science**. Jan. 1989.

10. T. Murata, V. S. Subrahmanian and T. Wakayama. *A Petri Net Model for Reasoning in the Presence of Inconsistency*. Submitted to: **IEEE Transactions on Data and Knowledge Engineering**, June 1989.

11. V. S. Subrahmanian. *Algebraic Foundations of Logic Programming, II: The Space of Multivalued and Paraconsistent Logic Programs*. Submitted to **Acta Informatica**. Feb. 1989.

12. V. S. Subrahmanian. *Approximate Reasoning in Logic Programming*. Submitted to **New Generation Computing**, March 1988.

13. V. S. Subrahmanian. *Y-Logic: A Framework for Reasoning about Chameleonic Programs with Inconsistent Completions*. Submitted to: **Fundamenta Informatica**, April 1989.

14. V. S. Subrahmanian. *Existential, Null and Partial Values in Disjunctive Deductive Databases*. Submitted to: **Journal of Automated Reasoning**, June 1989.

15. V. S. Subrahmanian. *Paraconsistent Disjunctive Deductive Databases*. Submitted to: **Theoretical Computer Science**, July 1989.

# *MISSION*
# *of*
# *Rome Air Development Center*

*RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence ($C^3I$) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of $C^3I$ systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.*